CodeWarrior®

C Compilers Reference

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Revised: 98/08/24 map

Metrowerks CodeWarrior copyright ©1993–1998 by Metrowerks Inc. and its licensors. All rights reserved.

Documentation stored on the compact disk(s) may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo, CodeWarrior, and Software at Work are registered trademarks of Metrowerks Inc. PowerPlant and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks and registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK(S) ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 9801 Metric, Suite 100 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Ordering	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
CompuServe	goto `Metrowerks`

Table of Contents

1 Introduction **11**

Read the Release Notes! 12
What's New 12
Conventions Used in This Manual 12

2 Setting C/C++ Compiler Options **15**

Setting C Compiler Options Overview 15
C/C++ Language Panel 15
C/C++ Warnings Panel 18
Treat All Warnings as Errors 20
Illegal Pragmas 20
Empty Declarations. 20
Possible Errors 21
Unused Variables. 22
Unused Arguments 23
Extra Commas 24
Extended Error Checking 24
Hidden Virtual Functions 26
Implicit Arithmetic Conversions 26
Non-Inlined Functions 26
Inconsistent Use of 'class' and 'struct' Keywords 27

3 C Compiler **29**

C Compiler Overview 29
The CodeWarrior Implementation of C 30
Identifiers 30
Include Files 30
Prefix Files 32
Sizeof() Operator 32
Volatile Variables 33
Enumerated Types 33
Extensions to ANSI C 36
ANSI Strict 37
Using the wchar_t Type 38

C++ Style Comments 38
Unnamed Arguments in Function Definitions 38
A # not Followed by Argument in a Macro. 38
Using an Identifier After #endif 39
Using Typecasted Pointers as lvalues 40
Declaring Variables By Address 40
ANSI Keywords Only. 41
Expand Trigraphs 42
Multibyte Character Constants 42
Inlining 43
Multibyte Strings and Comments. 44
Pool Strings 45
Reusing Strings 46
Require Function Prototypes. 47
Map Newlines to CR 49
Relaxed Pointer Type Rules 50
Use Unsigned Chars 50
Using 64-bit Integers 51
Converting Pointers to Types of the Same Size 51
Getting Alignment and Type Information at Compile-Time . 52
Arrays of Zero Length in Structures. 52

4 C++ Compiler 53

C++ Compiler Overview 53
CodeWarrior Implementation of C++ 54
Implicit Return Statement for main() 55
Keyword Ordering 55
Additional Keywords 56
Conversions in the Conditional Operator 56
Default Arguments in Member Functions 56
Local Class Declarations with Inline Functions 57
Copying and Constructing Class Objects 57
Checking for Resources To Initialize Static Data 58
Calling an Inherited Member Function 59
Unsupported Extensions. 61

Controlling the C++ Compiler 62
Using the C++ Compiler Always 62
Controlling ARM Conformance 63
Controlling Exception Handling 64
Controlling RTTI 64
Using the bool Type 65
Controlling C++ Extensions 65
Working With C++ Exceptions 66
Working With RTTI 67
Using the dynamic_cast Operator 67
Using the typeid Operator 69
Working With Templates. 70
Declaring and Defining Templates 70
Instantiating a Template 72

5 C++ and Embedded Systems 75

C++ and Embedded Systems Overview 75
Activating EC++ 75
Differences Between ANSI/ISO C++ and EC++ 76
Templates 76
Libraries 76
File Operations. 76
Localization 76
Exception Handling 77
Other Language Features 77
Meeting EC++ Specifications With CodeWarrior 77
Language Related Issues 77
Library Related Issues 78
Strategies for Smaller Code Size in C++ 78
Size Optimizations 79
Inlining 80
Virtual Functions 80
Runtime Type Identification 81
Exception Handling 81
Operator New 81

Multiple Inheritance 82
Virtual Inheritance 82
Stream-Based Classes 82
Alternative Class Libraries. 83

6 Pragmas and Symbols 85

Pragmas and Symbols Overview 85
Pragmas 86
Pragma Syntax 88
Pragma Scope 89
a6frames 90
align 91
align_array_members 92
always_inline 93
ANSI_strict 94
arg_dep_lookup 95
ARM_conform 95
auto_inline 96
bool 97
check_header_flags 98
code_seg 98
code68020 99
code68881 100
cplusplus 101
cpp_extensions. 101
d0_pointers 102
data_seg 104
def_inherited 104
defer_codegen 105
define_section 106
direct_destruction 110
direct_to_som 111
disable_registers 112
dollar_identifiers 112
dont_inline 113

dont_reuse_strings 114
ecplusplus. 114
EIPC_EIPSW 115
enumsalwaysint 115
exceptions. 116
export 117
extended_errorcheck 119
far_code, near_code, smart_code 121
far_data. 121
far_strings. 122
far_vtables 123
faster_pch_gen. 123
force_active 124
fourbyteints 124
fp_contract 125
fp_pilot_traps 126
function. 126
global_optimizer, optimization_level 127
IEEEdoubles. 127
ignore_oldstyle 128
import 129
init_seg 131
inline_depth. 131
inline_intrinsics 132
internal 132
interrupt 134
k63d 134
k63d_calls. 135
lib_export 135
longlong 136
longlong_enums 136
macsbug, oldstyle_symbols 137
mark 138
microsoft_exceptions 139
microsoft_RTTI 139

mmx 140
mmx_call 140
mpwc. 141
mpwc_newline. 142
mpwc_relax 143
no_register_coloring 143
once 145
only_std_keywords. 145
opt_common_subs 146
opt_dead_assignments 146
opt_dead_code. 147
opt_lifetimes. 147
opt_loop_invariants 148
opt_propagation 148
opt_strength_reduction 149
opt_unroll_loops 149
opt_vectorize_loops 150
optimization_level 150
optimize_for_size 150
oldstyle_symbols. 151
pack 151
parameter 152
pcrelstrings 153
peephole 154
pointers_in_A0, pointers_in_D0 155
pool_data 156
pool_strings 157
pop, push 157
precompile_target 158
profile 159
readonly_strings 160
register_coloring 161
require_prototypes 161
RTTI 162
scheduling 162

section 163
segment. 170
side_effects 171
simple_prepdump 172
SOMCallOptimization 172
SOMCallStyle 173
SOMCheckEnvironment 173
SOMClassVersion 175
SOMMetaClass 176
SOMReleaseOrder 176
stack_cleanup 177
static_inlines. 178
suppress_init_code 178
sym 179
syspath_once 180
toc_data. 180
trigraphs 181
traceback 181
unsigned_char 182
unused 183
use_fp_instructions. 183
use_frame 184
use_mask_registers 184
warn_emptydecl 185
warning_errors 185
warn_extracomma 186
warn_hidevirtual. 186
warn_illpragma 187
warn_implicitconv 188
warn_notinlined 189
warn_possunwant 189
warn_structclass 190
warn_unusedarg 191
warn_unusedvar 192
warning. 192

wchar_type 193
Predefined Symbols 193
ANSI Predefined Symbols 194
Metrowerks Predefined Symbols 195
Checking Options 198

Index **207**

1 Introduction

This manual describes the CodeWarrior C and C++ compilers, and how to use them to generate code for all CodeWarrior targets. There are chapters that cover information about the C/C++ compiler that applies to all targets. Information that is specific to a particular target operating system or processor appears in separate chapters, one per target.

The manual is organized into these principle sections:

- Interface—how to set compiler options
- Language—generic information on the compilers as they apply to all CW targets
- Pragmas—information on all pragmas for all targets

Each chapter begins with an overview section. Table 1.1 lists each of these and describes what each chapter in this manual covers.

Table 1.1 What's in this manual

This chapter…	Describes…
Setting C Compiler Options Overview	where to find information on the C/C++ Language and C/C++ Warnings settings panels.
C Compiler Overview	how the CodeWarrior C/C++ compiler implements C
C++ Compiler Overview	how the CodeWarrior C/C++ compiler implements C++ features that are not shared by C.
C++ and Embedded Systems Overview	how to use CodeWarrior C++ for embedded systems development and how to design programs to anticipate the proposed EC++ (Embedded C++) standard.
Pragmas and Symbols Overview	the pragmas and predefined symbols available with CodeWarrior C/C++ compiler

Read the Release Notes!

Because of last-minute changes to the CodeWarrior C/C++ compiler, some of the information may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

What's New

This reference has been updated to cover CodeWarrior C/C++ version 2.2 and later.

New and changed topics in this manual are:

- "Inconsistent Use of 'class' and 'struct' Keywords" on page 27—new option in the **C/C++ Warnings** settings panel
- "Converting Pointers to Types of the Same Size" on page 51—a new extension to the ANSI C standard
- "Getting Alignment and Type Information at Compile-Time" on page 52—new built-in functions
- "Arrays of Zero Length in Structures" on page 52—using arrays with no length in `structs`.
- "Pragmas" on page 86—many new (and previously undocumented) pragmas have been documented and this section has been reorganized into a format that's easier to use
- "Checking Options" on page 198—updated to cover new options

Conventions Used in This Manual

This manual includes syntax examples that describe how to use certain statements, such as the following:

```
#pragma parameter [return-reg] func-name [param-regs]
#pragma optimize_for_size on | off | reset
```

Table 1.2 describes how to interpret these statements.

Table 1.2 Understanding Syntax Examples

If the text looks like…	Then…
literal	Include it in your statement exactly as it's printed.
metasymbol	Replace the symbol with an appropriate value. The text after the syntax example describes what the appropriate values are.
a | b | c	Use one and only one of the symbols in the statement: either a, b, or c.
[a]	Include this symbol only if necessary. The text after the syntax example describes when to include it.

2

Setting C/C++ Compiler Options

This chapter describes where to find information on the C/C++ Compiler and C/C++ Warnings settings panels.

Setting C Compiler Options Overview

This section contains the following sections:

- "C/C++ Language Panel" on page 15, illustrates each option available to you, and tells you where that option is explained fully.
- "C/C++ Warnings Panel" on page 18, illustrates each compiler warning available to you, and tells you where that warning is explained fully.

The C/C++ Compiler settings panel, where you set compiler options, is also known as the C/C++ Language settings panel.

C/C++ Language Panel

You may configure how the C/C++ compiler works by setting a variety of options. You set these options in the C/C++ Compiler settings panel, shown in Figure 2.1. For information on how to display a particular settings panel, see the *IDE User Guide*.

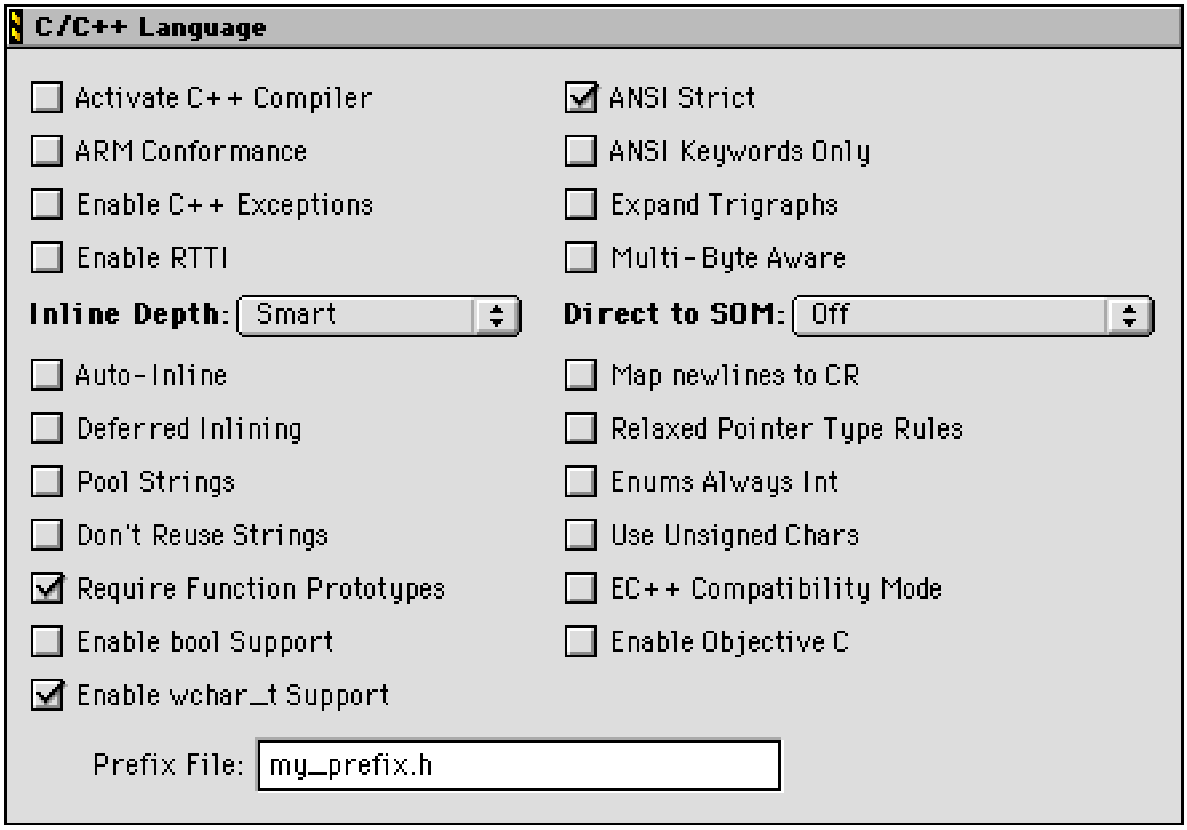
TIP: Another way to set these options is to use pragma directives in your source code. See "Pragmas and Symbols" on page 85 for more information.

Each compiler option has a corresponding pragma that you can use in source code to turn that particular option on or off, regardless of the settings in the C/C++ Compiler panel. In addition, there are some pragmas that do not have a corresponding setting in the C/C++ Compiler panel. See "Pragmas and Symbols Overview" on page 85 for details on each available pragma.

You may also use a special preprocessor directive in your code to determine the current setting of each option. See "Checking Options" on page 198 for information on how to use this directive.

Some of the options shown in Figure 2.1 may not appear for some targets. For example, the Direct to SOM item appears for Mac OS.

Figure 2.1 The C/C++ Compiler Settings Panel



Most of the items in this panel are discussed elsewhere in this manual, because they are closely related to how the compiler implements standard C and C++.

Other items

This table lists where to find information about the other items in this panel.

This item…	Is described here…
Activate C++ Compiler	"Using the C++ Compiler Always" on page 62.
ARM Conformance	"Controlling ARM Conformance" on page 63
Enable C++ Exceptions	"Controlling C++ Extensions" on page 65
Enable RTTI	"Controlling RTTI" on page 64
Inline Depth **Auto-inline**	"Inlining" on page 43
Pool Strings	"Pool Strings" on page 45
Don't Reuse Strings	"Reusing Strings" on page 46
Require Function Prototypes	"Require Function Prototypes" on page 47
Enable bool Support	"Using the bool Type" on page 65
Enable wchar_t Support	"Using the wchar_t Type" on page 38
ANSI Strict	"ANSI Strict" on page 37
ANSI Keywords Only	"ANSI Keywords Only" on page 41
Expand Trigraphs	"Expand Trigraphs" on page 42
Multi-Byte Aware	"Multibyte Strings and Comments" on page 44
Direct to SOM	*Targeting Mac OS* manual
Map Newlines to CR	"Map Newlines to CR" on page 49
Relaxed Pointer Type Rules	"Relaxed Pointer Type Rules" on page 50
Enums Always Int	"Enumerated Types" on page 33

This item…	Is described here…
Use Unsigned Chars	"Use Unsigned Chars" on page 50
Prefix File	"Prefix Files" on page 32
EC++ Compatibility Mode	"Activating EC++" on page 75
Enable Objective C	*Targeting Rhapsody* manual

C/C++ Warnings Panel

The C/C++ compiler generates errors when it cannot understand your code as a result of improper syntax. In addition to errors, the compiler can generate a series of helpful warnings that alert you to legal but usually unintentional syntax. These mistakes are legal C and C++ code, but the code might not do what you expect.

When the compiler finds one of these possible mistakes, it can generate a warning. Warnings are not fatal. Unless you choose to treat these warnings as errors, your code will still compile (although it may not run correctly).

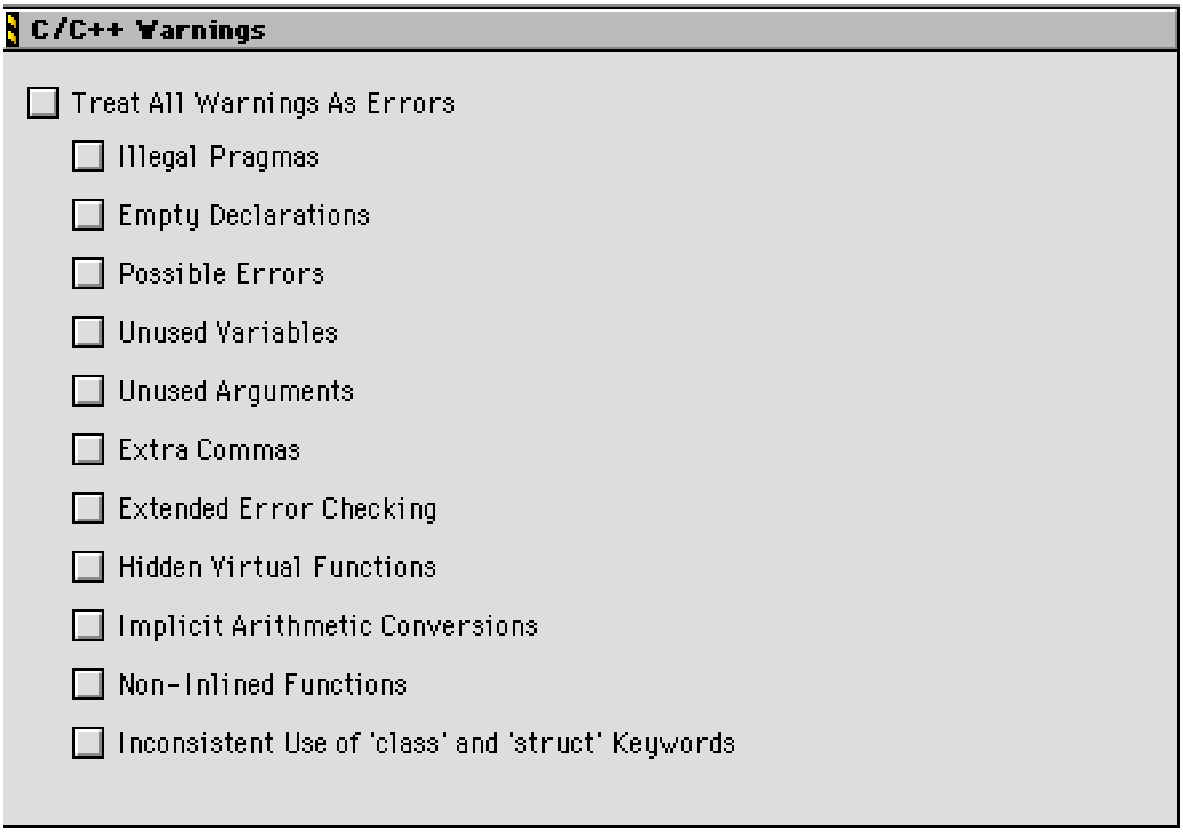
This section describes the items in the C/C++ Warnings panel. You determine which warnings you receive by setting options in the panel. Figure 2.2 illustrates this panel.

Each warning also has a corresponding pragma that you can use in source code to turn that particular warning on or off for a limited piece of code. See "Pragmas and Symbols Overview" on page 85 for details on each available pragma.

You may also use a special preprocessor directive in your code to determine the current setting of each option. See "Checking Options" on page 198 for information on how to use this directive.

For information on how to display a particular settings panel, see the *IDE User Guide*.

Figure 2.2 The C/C++ Warnings Settings Panel



The items in this panel are:

- "Treat All Warnings as Errors" on page 20
- "Illegal Pragmas" on page 20
- "Empty Declarations" on page 20
- "Possible Errors" on page 21
- "Unused Variables" on page 22
- "Unused Arguments" on page 23
- "Extra Commas" on page 24
- "Extended Error Checking" on page 24
- "Hidden Virtual Functions" on page 26
- "Implicit Arithmetic Conversions" on page 26

- "Non-Inlined Functions" on page 26
- "Inconsistent Use of 'class' and 'struct' Keywords" on page 27

Treat All Warnings as Errors

When the **Treat All Warnings as Errors** option is on, the compiler treats all warnings as though they were errors. It will not compile a file until all warnings are resolved.

The **Treat All Warnings as Errors** option corresponds to the pragma `warning_errors`, described at"warning_errors" on page 185. To check whether this option is on, use `__option (warning_errors)`.See "Checking Options" on page 198 for information on how to use this directive.

Illegal Pragmas

If the **Illegal Pragmas** option is on, the compiler displays a warning when it encounters an illegal pragma. For example, these pragma statements generate warnings:

```
#pragma near_data off    // WARNING: near_data is not a pragma.
#pragma far_data select  // WARNING: select is not defined
#pragma far_data on      // OK
```

The **Illegal Pragmas** option corresponds to the pragma `warn_illpragma`, described at "warn_illpragma" on page 187. To check whether this option is on, use `__option (warn_illpragma)`.See "Checking Options" on page 198 for information on how to use this directive.

Empty Declarations

If the **Empty Declarations** option is on, the compiler displays a warning when it encounters a declaration with no variable name. For example:

```
int ;          // WARNING
int i;         // OK
```

The **Empty Declarations** option corresponds to the pragma `warn_emptydecl`, described at "warn_emptydecl" on page 185. To check whether this option is on, use `__option (warn_emptydecl)`. See "Checking Options" on page 198 for information on how to use this directive.

Possible Errors

If the **Possible Errors** option is on, the compiler checks for some common typographical mistakes that are legal C syntax but that may have unwanted side effects, such as putting in unintended semicolons or confusing = and ==. The compiler generates a warning if it encounters one of these:

- An assignment in a logical expression or the condition in an `if`, `while`, or `for` expression. This check is useful if you frequently use = when you meant to use ==. For example:

```
if (a=b) f();          // WARNING: a=b is an assignment

if ((a=b)!=0) f();     // OK: (a=b)!=0 is a comparison

if (a==b) f();         // OK: (a==b) is a comparison
```

- An equal comparison in a statement that contains a single expression. This check is useful if you frequently use == when you meant to use =. For example:

```
a == 0;          // WARNING: This is a comparison.
a = 0;           // OK: This is an assignment
```

- A semicolon (`;`) directly after a `while`, `if`, or `for` statement. For example, the statement generates a warning and is probably an unintended infinite loop:

```
while (i++);  // WARNING: Unintended infinite loop
```

If you intended to create an infinite loop, put white space or a comment between the `while` statement and the semicolon. For example, these statements do not generate errors or warnings, and ensure that you will win the hearts of everyone using your software when the code is executed.

```
while (i++) ; // OK: White space separation
while (i++)   /*: Comment separation */ ;
```

The **Possible Errors** option corresponds to the pragma `warn_possunwant`, described at "warn_possunwant" on page 189. To check whether this option is on, use `__option (warn_possunwant)`. See "Checking Options" on page 198 for information on how to use this directive.

Unused Variables

If the **Unused Variables** option is on, the compiler generates a warning when it encounters a variable that you declare but do not use. This check helps you find misspelled variable names and variables you have written out of your program. For example:

```
void foo(void)
{
  int temp, errer;       // ERROR: errer is misspelled
  error = do_something() // WARNING: temp and error are unused.
}
```

If you want to use this warning, but need to declare a variable that you don't use, use the pragma `unused`, as in this example:

```
void foo(void)
{
  int i, temp, error;

  #pragma unused (i, temp) /* Compiler won't warn that i and temp
```

```
  error=do_something();    /* are not used */

}
```

The **Unused Variables** option corresponds to the pragma `warn_unusedvar`, described at "warn_unusedvar" on page 192. To check whether this option is on, use `__option (warn_unusedvar)`. See "Checking Options" on page 198 for information on how to use this directive.

Unused Arguments

If the **Unused Arguments** option is on, the compiler generates a warning when it encounters an argument you declare but do not use. This check helps you find misspelled argument names and arguments you have written out of your program.

```
void foo(int temp,int errer); // ERROR: errer is misspelled
{
  error = do_something(); // WARNING: temp and error are unused.
}
```

If you need to declare an argument that you don't use, there are two ways to avoid this warning. You can use the pragma `unused`, as in this example:

```
void foo(int temp, int error)
{
  #pragma unused (temp)
  /* Compiler won't warn that temp is not used */

  error=do_something();
}
```

You can also turn off the ANSI Strict option, and not give the unused argument a name. (See "Unnamed Arguments in Function Definitions" on page 38.) For example:

```
void foo(int /* temp */, int error)
{
  /* Compiler won't warn that temp is not used, it's not named

  error=do_something(); */
}
```

The **Unused Arguments** option corresponds to the pragma `warn_unusedarg`, described at "warn_unusedarg" on page 191. To check whether this option is on, use `__option (warn_unusedarg)`. See "Checking Options" on page 198 for information on how to use this directive.

Extra Commas

If the **Extra Commas** option is on, the compiler generates a warning when it encounters an extra comma. For example, this statement is legal in C, but it causes a warning when this option is on:

```
int a[] = { 1, 2, 3, 4, }; // ^ WARNING: Extra comma after 4
```

The **Extra Commas** option corresponds to the pragma `warn_extracomma`, described at "warn_extracomma" on page 186. To check whether this option is on, use `__option (warn_extracomma)`. See "Checking Options" on page 198 for information on how to use this directive.

Extended Error Checking

If the **Extended Error Checking** option is on, the C compiler generates a warning (not an error) if it encounters one of these syntax problems:

- A non-`void` function that does not contain a `return` statement. For example, this would generate a warning:

```
main()        /* assumed to return int */
{
```

```
  printf ("hello world\n");
} /* WARNING: no return statement */
```

This would be OK:

```
void main() /* function declared to return void */
{
  printf ("hello world\n");
}
```

- Assigning an integer or floating-point value to an `enum` type. For example:

```
enum Day { Sunday, Monday, Tuesday, Wednesday,
           Thursday, Friday, Saturday } d;

d = 5;             /* WARNING */
d = Monday;        /* OK */
d = (Day)3  ;      /* OK */
```

- An empty `return` statement (`return;`) in a function that is not declared `void`. For example, this code would generate a warning:

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err !=0) return;  /* ERROR: Empty return statement */

  /* ... */
```

This would be OK:

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err!=0) return -1; /* OK */

  /* ... */
```

The **Extended Error Checking** option corresponds to the pragma `extended_errorcheck`, described at "extended_errorcheck" on page 119. To check whether this option is on, use `__option (extended_errorcheck)`. See "Checking Options" on page 198 for information on how to use this directive.

Hidden Virtual Functions

If the **Hidden virtual functions** option is on, the compiler generates a warning if you declare a non-virtual member function in a subclass that hides an inherited virtual function in a superclass. One function hides another if it has the same name but a different argument types. For example:

```
class A {
  public:
    virtual void f(int);
    virtual void g(int);
};

class B: public A {
  public:
    void f(char);        // WARNING: Hides A::f(int)
    virtual void g(int); // OK: Overrides A::g(int)
};
```

The **Hidden virtual functions** option corresponds to the pragma `warn_hidevirtual`, described at "warn_hidevirtual" on page 186. To check whether this option is on, use `__option (warn_hidevirtual)`. See "Checking Options" on page 198 for information on how to use this directive.

Implicit Arithmetic Conversions

If the **Implicit Arithmetic Conversions** option is on, the compiler issues a warning if the destination of an operation isn't large enough to hold all possible results. For example, assigning the value of a variable of type `long` to a variable of type `char` will result in a warning if this option is on.

Non-Inlined Functions

If the **Non-Inlined Functions** option is on, the compiler issues a warning when it is unable to inline a function.

This option corresponds to pragma `warn_notinlined`. To check whether this option is on, use `__option (warn_notinlined)`. See "Checking Options" on page 198 for information on how to use this directive.

Inconsistent Use of 'class' and 'struct' Keywords

If the **Inconsistent Use of 'class' and 'struct' Keywords** option is one, the compiler issues a warning if the `class` and `struct` keywords are used in the definition and declaration of the same identifier.

```
class X;
struct X { int a; }; // warning
```

Use this warning when linking (with static or dynamic libraries) with object code produced from another C++ compiler that makes a distinction between class and struct variables in its name "mangling."

This option corresponds to pragma `warn_structclass`. To check whether this option is on, use `__option (warn_structclass)`. See "Checking Options" on page 198 for information on how to use this directive.

3

C Compiler

This chapter describes how the CodeWarrior C compiler implements the C language.

C Compiler Overview

This chapter discusses the CodeWarrior C compiler as it applies to all CodeWarrior targets. For the most part, the information in this chapter is equally applicable to any operating system or processor.

Other chapters in this manual discuss other features of the compiler that are specific to particular operating systems and processors. For a complete picture, you need to consider all the information relating your target of interest.

This chapter does not cover C++ features. For more information on the C++ language, see "C++ Compiler Overview" on page 53.

This chapter contains the following sections:

- "The CodeWarrior Implementation of C" on page 30 explains how the CodeWarrior compiler implements certain parts of the standard C language.
- "Extensions to ANSI C" on page 36 describe some of CodeWarrior C's extensions to the C standards.

You'll find frequent references to K&R §A, which is Appendix A, "Reference Manual," of *The C Programming Language, Second Edition* (Prentice Hall) by Kernighan and Ritchie. These references show you where to look for more information on the topics discussed in the corresponding section.

The CodeWarrior Implementation of C

This section describes how CodeWarrior implements many parts of the C programming language. For information on the parts of the C++ language that are specific to C++, see "C++ Compiler Overview" on page 53.

This section discusses the following topics:

- Identifiers
- Include Files
- Prefix Files
- Sizeof() Operator
- Volatile Variables
- Enumerated Types

Identifiers

(K&R, §A2.3) The C compiler lets let you create identifiers of any size. However, only the first 255 characters are significant for internal and external linkage.

Include Files

(K&R, §A12.4) The C compiler can nest `#include` files up to 32 times. An include file is nested if another `#include` file uses it in an `#include` statement. For example, if `Main.c` includes the file `MyFunctions.h`, which includes the file `MyUtilities.h`, the file `MyUtilities.h` is nested once.

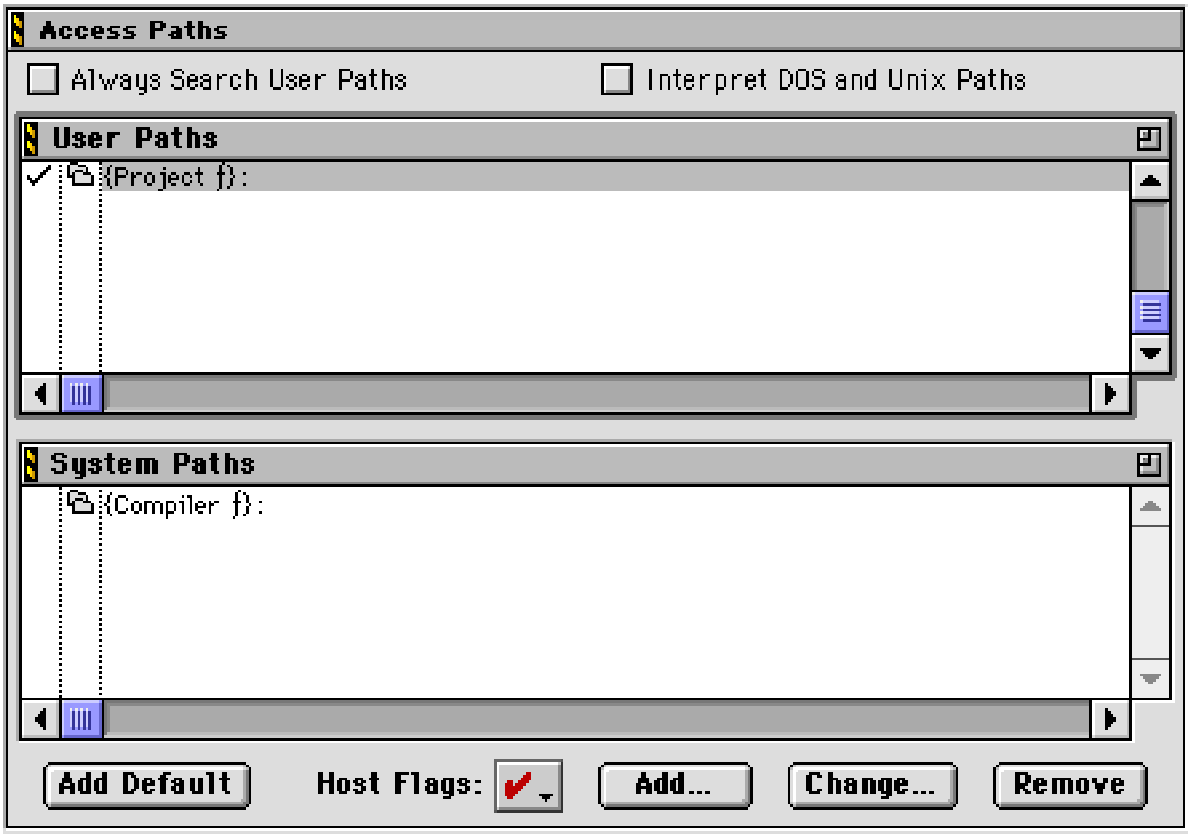
You can use full path names in `#include` directives, as in this example:

```
#include "HD:Tools:my headers:macros.h"
```

The CodeWarrior IDE lets you specify where the compiler looks for `#include` files with the Access Paths settings panel, shown in Figure 3.1. It contains two lists of folders: the User list and the System list. By default, each list contains one folder. The User list contains `{Project ƒ}`, which is the folder that the project file is in and all

the folders it contains. The System list contains `{Compiler ƒ}`, which is the folder that CodeWarrior is in, and all the folders it contains.

Figure 3.1 The Access Paths settings panel



The compiler searches for a `#include` file in either the System list or both the User and System lists, depending on which characters enclose the file. If you enclose the file in brackets (`#include <stdio.h>`), the compiler looks for the file in the System list. If you enclose the file in quotes (`#include "myfuncs.h"`), the compiler looks for the file in the User list first, and then in the System list. In general, use brackets for include files that are for a large variety of projects and use quotes for include files that are for a specific project.

If the **Always Search User Paths** option is selected, the compiler also uses the User access paths for include files enclosed in brackets (`#include <`*file*`>`).

For information on how to modify the Access Paths settings panel, see the *CodeWarrior IDE User Guide*.

See also: "Prefix Files" on page 32.

TIP: If you're using the compiler under Apple Computer's MPW programming environment, you can specify where to find #include files with the `-i` compiler option and the `{CIncludes}` variable, described in Command-Line Tools Manual and MPW Command Reference.

Prefix Files

If you have a single file you wish to include in every source file in a project, you can use the **Prefix File** item in the C/C++ Language Panel. Enter the name of the file in the Prefix File edit field in this panel.

The compiler automatically includes this file (and any files that it, in turn, includes) in every source file in the project's current target. This is an excellent way to include a precompiled header file in a project.

See also: "Include Files" on page 30.

Sizeof() Operator

The `sizeof()` operator returns a number of type `size_t`, which the compiler declares to be of type `unsigned long int` (in the file `stddef.h`). If your code assumes that `sizeof()` returns a number of type `int`, it may not work correctly.

Volatile Variables

(K&R, §A4.4) When you declare a variable to be volatile the C compiler takes the following precautions:

- It does not store the variable in a register.
- It computes the variable's address every time a piece of code references the variable.

Listing 3.1 shows an example of volatile variables.

Listing 3.1 volatile variables

```
void main(void)
{
    int i[100];
    volatile int a, b;

    a = 5;
    b = 20;

    i[a + b] = 15;
    i[a + b] = 30;
}
```

The compiler does not place the value of `a`, `b`, or `a+b` in registers. Also, the compiler recalculates a+b in both assignment statements.

Enumerated Types

(K&R, §A8.4) This section describes how the C compiler selects the underlying integer data type for an enumerated type. The compiler behavior is controlled primarily by the **Enums Always Int** option in the C/C++ Language Panel.

The compiler uses one of two different strategies, depending on the setting of the Enums Always Int option.

If Enums Always Int is *on*, the underlying type is always `signed int`. All enumerators must be no larger than a `signed int`. If an

enumerated constant is larger than an `int`, the compiler generates an error.

However, if the **ANSI Strict** option is off, enumerators that can be represented as an `unsigned int` are implicitly converted to `signed int`. For example:

```
#pragma enumsalwaysint on
#pragma ANSI_strict on
enum foo { a=0xFFFFFFFF }; // ERROR. a is 4,294,967,295:
                                                      //
too big for a signed int
#pragma ANSI_strict off
enum bar { b=0xFFFFFFFF }; // OK: b can be represented as an
                                                        //
unsigned int, but is implicitly
                                                        //
converted to a signed int (-1).
```

See "ANSI Strict" on page 37 for additional features of that setting.

If Enums Always Int is *off*, the compiler chooses the integral data type that supports the largest enumerated constant. The type could be as small as a `char` or as large as a `long int`. It could even be a 64-bit `long long` value.

To be more precise, if Enums Always Int is off, the compiler picks one of the following:

- If all enumerators are positive, it picks the smallest unsigned integral base type that is large enough to represent all enumerators
- If at least one enumerator is negative, it picks the smallest signed integral base type large enough to represent all enumerators.

For example:

```
#pragma enumsalwaysint off
enum { a=0,b=1 };          // base type: unsigned char
```

```
enum { c=0,d=-1 };         // base type: signed char
enum { e=0,f=128,g=-1 };   // base type: signed short
```

The compiler will use `long long` data types only if Enums Always Int is off and the `longlong_enums` pragma is on. (There is no settings panel option corresponding to the `longlong_enums` pragma.)

For example:

```
#pragma enumsalwaysint off
#pragma longlong_enums off
enum { a=0x7FFFFFFFFFFFFFFF }; // ERROR: a is too large
#pragma longlong_enums on
enum { b=0x7FFFFFFFFFFFFFFF };// OK: base type: signed long long
enum { c=0x8000000000000000 };// OK: base type: unsigned long long
enum { d=-1,e=0x80000000 };     // OK: base type: signed long long
```

When the `longlong_enums` pragma is off and **ANSI Strict** is on, you cannot mix unsigned 32-bit enumerators greater than `0x7FFFFFFF` and negative enumerators. If both the `longlong_enums` pragma and the ANSI Strict option are off, huge unsigned 32-bit enumerators are implicitly converted to signed 32-bit types.

For example:

```
#pragma enumsalwaysint off
#pragma longlong_enums off
#pragma ANSI_strict on
enum { a=-1,b=0xFFFFFFFF };    // error
#pragma ANSI_strict off
enum { c=-1,d=0xFFFFFFFF };    // base type: signed int (b==-1)
```

The **Enums Always Int** option corresponds to the pragma `enumsalwaysint`. To check whether this option is on, use `__option(enumsalwaysint)`. By default, this option is off.

See also "enumsalwaysint" on page 115, "longlong_enums" on page 136, and "Checking Options" on page 198.

Extensions to ANSI C

This section describes CodeWarrior extensions to the C standard that apply to all targets. In most cases you turn the extension on or off with an option in the C/C++ Language Panel. See "C/C++ Language Panel" on page 15 for information about that panel.

The topics in this section are:

- "ANSI Strict" on page 37
- "C++ Style Comments" on page 38
- "Unnamed Arguments in Function Definitions" on page 38
- "A # not Followed by Argument in a Macro" on page 38
- "Using an Identifier After #endif" on page 39
- "Using Typecasted Pointers as lvalues" on page 40
- "Declaring Variables By Address" on page 40
- "ANSI Keywords Only" on page 41
- "Expand Trigraphs" on page 42
- "Multibyte Character Constants" on page 42
- "Inlining" on page 43
- "Multibyte Strings and Comments" on page 44
- "Reusing Strings" on page 46
- "Require Function Prototypes" on page 47
- "Map Newlines to CR" on page 49
- "Relaxed Pointer Type Rules" on page 50
- "Use Unsigned Chars" on page 50
- "Using 64-bit Integers" on page 51
- "Converting Pointers to Types of the Same Size" on page 51
- "Getting Alignment and Type Information at Compile-Time" on page 52
- "Arrays of Zero Length in Structures" on page 52

For information on target-specific extensions, you should refer to the Target manual for the particular target in which you are interested.

ANSI Strict

The ANSI Strict option in the C/C++ Language Panel affects several extensions to the C language supported by the CodeWarrior compiler. The extensions are:

- C++ Style Comments
- Unnamed Arguments in Function Definitions
- A # not Followed by Argument in a Macro
- Using an Identifier After #endif
- Using Typecasted Pointers as lvalues
- Converting Pointers to Types of the Same Size
- Arrays of Zero Length in Structures

In each case the extension is only available if the ANSI Strict setting is *off*. If the ANSI Strict setting is on, then these extensions to the ANSI C standard are disabled.

You cannot turn on the extensions controlled by the ANSI Strict setting individually. They are all on or off depending upon the setting.

This setting may affect how the compiler handles enumerated constants. See "Enumerated Types" on page 33 for more information.

This setting may also affects the `main()` function for C++ programs. See "Implicit Return Statement for main()" on page 55..

The ANSI Strict option corresponds to the pragma ANSI_strict. To check whether this option is on, use `__option (ANSI_strict)`.

See also "ANSI_strict" on page 94. and "Checking Options" on page 198.

Using the wchar_t Type

Turn on the **Enable wchar_t Support** option if you want to use the standard C++ `wchar_t` type to represent wide characters. Turn this option off to use the regular character type, `char`.

C++ Style Comments

(K&R, §A2.2) The C compiler can accept C++ comments in source code. In a C++ comment, anything that follows `//` on a line is considered a comment. For example:

```
a = b; // This is a C++ comment
```

To turn this feature on, turn *off* the ANSI Strict setting in the C/C++ Language Panel. If the ANSI Strict setting is on, then this extension to standard C is disabled.

See also: "ANSI Strict" on page 37.

Unnamed Arguments in Function Definitions

(K&R, §A10.1) The C compiler can accept an unnamed argument in a function definition. For example:

```
void f(int ) {}   /* OK, if ANSI Strict is off */
void f(int i) {}  /* ALWAYS OK                                  */
```

To turn this feature on, turn *off* the ANSI Strict setting in the C/C++ Language Panel. If the ANSI Strict setting is on, then this extension to standard C is disabled.

See also: "ANSI Strict" on page 37.

A # not Followed by Argument in a Macro

(K&R, §A12.3) The C compiler can accept a # token not followed by an argument in a macro definition. For example:

```
#define add1(x) #x #1  // OK, but probably not what you wanted:
                       // add1(abc) creates "abc"#1
#define add2(x) #x "2" // OK: add2(abc) creates "abc2"
```

To turn this feature on, turn *off* the ANSI Strict setting in the C/C++ Language Panel. If the ANSI Strict setting is on, then this extension to standard C is disabled.

See also: "ANSI Strict" on page 37.

Using an Identifier After #endif

(K&R, §A12.5) The C compiler can accept an identifier token after `#endif` and `#else`. This extension helps you match an `#endif` statement with its corresponding `#if`, `#ifdef`, or `#ifndef` statement, as shown here:

```
#ifdef __MWERKS__
#    ifndef __cplusplus
        /*
         * . . .
         */
#    endif __cplusplus
#endif __MWERKS__
```

To turn this feature on, turn *off* the ANSI Strict setting in the C/C++ Language Panel. If the ANSI Strict setting is on, then this extension to standard C is disabled.

See also: "ANSI Strict" on page 37.

TIP: If you turn on the **ANSI Strict** option (disabling the extension), you can still use the same idea to help you match your `#ifdef` and `#endif` directives. Simply put the identifiers into comments, as in the following example.

```
#ifdef __MWERKS__
#    ifndef __cplusplus
        /*
         * . . .
        */
#    endif /* __cplusplus */
#endif /* __MWERKS__ */
```

Using Typecasted Pointers as lvalues

The C compiler can accept a pointer that you typecast to another pointer type as an lvalue.

For example:

```
char *cp;
((long *) cp)++;  /* OK if ANSI Strict is off. */
```

To turn this feature on, turn *off* the ANSI Strict setting in the C/C++ Language Panel. If the ANSI Strict setting is on, then this extension to standard C is disabled.

See also: "ANSI Strict" on page 37.

Declaring Variables By Address

(K&R, §A8.7) The C compiler lets you specify the address that a variable refers to. For example, this definition defines `MemErr` to contain whatever is at the address `0x0220`:

```
short MemErr:0x220;
```

The variable `MemErr` contains whatever is at the address `0x220`.

WARNING! For Mac OS programming, avoid using this extension to refer to low-memory globals. To ensure that your programs

are compatible with future versions of the Mac OS, use the functions defined in the `LowMem.h` header file.

This extension cannot be turned off. There is no corresponding pragma or setting in the C/C++ Language Panel.

ANSI Keywords Only

(K&R, §A2.4) The C compiler can recognize several additional reserved keywords. The **ANSI Keywords Only** option in the C/C++ Language Panel controls whether the compiler recognizes these keywords.

If this option is *on*, the compiler generates an error if it encounters any of the CodeWarrior C additional keywords. If you're writing code that must follow the ANSI standard strictly, turn on the **ANSI Keywords Only** option in the settings panel.

When this option is *off*, these additional keywords are available to you:

- `asm`

  This keyword lets you use the compiler's built-in inline assembler. (K&R, §A10.1) For more information on the inline assemblers, consult:

- `far`

  This keyword is used in Mac OS programming.

- `inline`

  Lets you declare a C function to be inline. For more information, see "Inlining" on page 43.

- `pascal`

  This keyword is used in Mac OS programming.

- `__stdcall`

  This keyword is used in Microsoft Win32 programming.

The **ANSI Keywords Only** option corresponds to the pragma `only_std_keywords`. To check whether this option is on, use `__option (only_std_keywords)`. By default, this option is off.

See also "only_std_keywords" on page 145 and "Checking Options" on page 198.

Expand Trigraphs

(K&R, §A12.1) The C compiler lets you ignore trigraph characters. Many common character constants (especially on Mac OS) look like trigraph sequences, and this extension lets you use them without including escape characters.

If you're writing code that must follow the ANSI standard strictly, turn *on* the **Expand Trigraphs** option in the C/C++ Language Panel. If this option is on, be careful when you initialize strings or multicharacter constants that contain question marks.

```
char c = '????';          // ERROR:  Trigraph sequence expands to
'??^
char d = '\?\?\?\?';  // OK
```

The Expand Trigraphs option corresponds to the pragma `trigraphs`, To check whether this option is on, use `__option (trigraphs)`. By default, this option is off.

See also "trigraphs" on page 181 and "Checking Options" on page 198.

Multibyte Character Constants

(K&R, §A2.5.2) The C compiler lets you use multibyte character constants that contain 2 to 4 characters. Here are some examples:

Table 3.1 Multibyte character constant

Character constant	Equivalent hexadecimal
`'ABCD'`	0x41424344
`'ABC'`	0x00414243
`'AB'`	0x00004142

This extension cannot be turned off. There is no corresponding pragma or setting in the C/C++ Language Panel.

See "Multibyte Strings and Comments" on page 44 for a discussion of a different extension relating to using character sets with more than 256 characters (such as Kanji).

Inlining

The compiler determines whether to inline a function based on the settings of the **ANSI Keywords Only** item and the **Inline Depth** and **Auto-inline** items in the C/C++ Language Panel.

For beginners: When you call an inline function, the compiler inserts the function's code instead of a function call. Inlining functions makes your programs faster (because you execute the function's code immediately without a function call), but possibly larger (because the function's code may be repeated in several different places).

If you turn *off* the **ANSI Keywords Only** option, you can declare C functions to be `inline`. The inlining items in the C/C++ Language Panel let you choose to inline no functions, only functions declared inline, or all small functions as shown in Table 3.2.

Table 3.2 Options for the Inline Depth pop-up menu

This option	Does this…
Don't Inline	Inlines no functions, not even C or C++ functions declared `inline`.
Smart	Inlines small functions to a depth of 2 to 4 inline functions deep.
1 to **8**	Always inlines to the depth specified by the numerical selection.
Always Inline	Always inlines functions, no matter the depth.

The **Smart** item and items **1** to **8** in the **Inline Depth** pop-up menu correspond to the `pragma inline_depth` ("C/C++ Language Panel" on page 15). To check whether this option is on, use `__option(inline_depth)`, described at "Checking Options" on page 198.

The **Don't Inline** item in the Inline Depth pop-up menu corresponds to the pragma `dont_inline`, described at "dont_inline" on page 113. To check whether this option is on, use `__option (dont_inline)`, described at "dont_inline" on page 199. By default, this option is off.

The **Auto-Inline** option lets the compiler choose which functions to inline. Also inlines C++ functions declared `inline` and member functions defined within a class declaration. This option corresponds to the pragma `auto_inline`, described at "auto_inline" on page 96. To check whether this option is on, use `__option (auto_inline)`, described at "auto_inline" on page 199. By default, this option is off.

Multibyte Strings and Comments

The C compiler supports languages that use more than one byte to represent a character, such as Kanji. This feature is controlled by the **Multi-Byte Aware** item in the C/C++ Language Panel.

To use multibyte strings or comments, turn on the Multi-Byte Aware option. If you don't need multibyte strings or comments, turn this option off, because it slows down the compiler.

See "Multibyte Character Constants" on page 42 for a discussion of creating a character constant consisting of more than one character (a completely different subject with a similar sounding name).

Pool Strings

The **Pool Strings** option in the C/C++ Language Panel affects how the compiler stores string constants

NOTE: In principle this option works for all targets. However, it is useful only for a Table of Contents based (TOC-based) linking mechanism such as that used for Mac OS on the PowerPC processor, or with Code Fragment Manager support on the 68K processor.

If this option is on, the compiler collects all string constants into a single data object so your program needs one TOC entry for all of them. Turning this option on decreases the number of TOC entries in your program but increases your program's size, because it uses a less efficient method to store the string's address.

If this option is off, the compiler creates a unique data object and TOC entry for each string constant.

TIP: You can change the size of the TOC with the **Store Static Data in TOC** option in the PPC Processor settings panel. For more information, see the Targeting Mac OS manual.

Turning this option on is especially useful if your program is large and has many string constants.

NOTE: If you turn the **Pool Strings** option on, the compiler ignores the setting of the **PC-Relative Strings** option. This is a 68K-only feature.

The **Pool Strings** option corresponds to the pragma `pool_strings`. To check whether this option is on, use `__option (pool_strings)`. By default, this option is off.

See also "pool_strings" on page 157 and "Checking Options" on page 198.

Reusing Strings

The **Don't Reuse Strings** option in the C/C++ Language Panel affects how the compiler stores string literals.

If this option is on, the compiler stores each string literal separately.

If this option is off, the compiler stores only one copy of identical string literals. This option helps you save memory if your program contains identical string literals which you do not modify.

If this option is off (meaning that string storage is reused for identical strings) and you change one of the strings, you change them all. For example, take this code snippet:

```
char *str1="Hello";
char *str2="Hello"; // two identical strings
*str2 = 'Y';
```

If the Don't Reuse Strings option is *on*. the strings are stored separately. After changing the first character, `str1` is still `"Hello"` but `str2` is `"Yello"`.

If the Don't Reuse Strings option is *off*, the two strings are stored in one memory location (i.e. the same memory location is reused), because they are both identical. After changing the first character, *both* `str1` and `str2` are `"Yello"`. This is counter-intuitive, and can lead to difficult-to locate bugs.

The Don't Reuse Strings option corresponds to the pragma `dont_reuse_strings`. To check whether this option is on, use `__option (dont_reuse_strings)`. By default, this option is on. (Strings are *not* reused.)

See also "dont_reuse_strings" on page 114. and "Checking Options" on page 198.

Require Function Prototypes

(K&R, §A8.6.3, §A10.1) The C compiler lets you choose how to enforce function prototypes. This behavior is controlled by the **Require Function Prototypes** item in the C/C++ Language Panel.

When the **Require Function Prototypes** option is on, the compiler generates an error if you use a function that is defined after it is referenced and does not have a prototype. If the function is implicitly defined, that is, is defined before it is referenced, and does not have a prototype, then the compiler will issue a warning if **Require Function Prototypes** is on.

This option helps you prevent errors that happen when you call a function before you declare or define it. For example, without a function prototype, you may pass data if the wrong type. As a result, your code may not work as you expect even though it compiles without error.

In Listing 3.2, `PrintNum()` is called with an integer argument but is later defined to take a floating-point argument.

Listing 3.2 Unnoticed type-mismatch

```
#include <stdio.h>

void main(void)
{
  PrintNum(1);  // PrintNum() tries to interpret the
                   integer as a float. Prints 0.000000.
}
```

```
void PrintNum(float x)
{
    printf("%f\n", x);
}
```

When you run this code, you could get this result:

```
0.000000
```

Although the compiler does not complain about the type mismatch, the function does not work as you want. Since `PrintNum()` is not prototyped, the compiler does not know it needs to convert the integer to a floating-point number before calling the function. Instead, the function interprets the bits it received as a floating-point number and prints nonsense.

If you prototype `PrintNum()` first, as in Listing 3.3, the compiler converts its argument to a floating-point number, and the function prints what you wanted.

Listing 3.3 Using a prototype to avoid type-mismatch

```
#include <stdio.h>

void PrintNum(float x);  // Function prototype.

void main(void)
{
    PrintNum(1);    //  Compiler knows to convert int to float.
}                               //  Prints 1.000000.

void PrintNum(float x)
{
    printf("%f\n", x);
}
```

In the above example, the compiler automatically typecast the passed value. In other situations where automatic typecasting is not available, the compiler will generate an error if an argument does not match the data type required by a function prototype. Such a

mismatched data type error is easy to locate at compile time. If you do not use prototypes, you get no compiler error. However, at runtime the code may behave strangely, and the cause of the resulting unintentional behavior can be extremely difficult to track down.

The **Require Function Prototypes** option corresponds to the pragma `require_prototypes`. To check whether this option is on, use `__option (require_prototypes)`. By default, this option is on.

See also "require_prototypes" on page 161, and "Checking Options" on page 198.

Map Newlines to CR

The C compiler lets you choose how to interpret the newline (`'\n'`) and return (`'\r'`) characters. This behavior is controlled by the **Map Newlines to CR** item in the C/C++ Language Panel.

In most compilers, including CodeWarrior C/C++, `'\r'` is translated to the value `0x0D`, the standard value for carriage return, and `'\n'` is translated to the value `0x0A`, the standard value for linefeed.

However, the C compiler in the Macintosh Programmers Workshop, known as MPW C, `'\r'` is translated to `0x0A` and `'\n'` is translated to `0x0D`—the opposite of the typical behavior.

If you turn this option on, the compiler uses the MPW conventions for the `'\n'` and `'\r'` characters.

If this option is off, the compiler uses the CodeWarrior C and C++ conventions for these characters.

If you turn this option on, be sure you use ANSI C and C++ libraries that were compiled with this option on. If you turn this option on and use libraries built with this option off, you won't be able to read and write `'\n'` and `'\r'` properly. For example, printing '\n' would bring you to the beginning of the current line instead of inserting a new line.

This option corresponds to the pragma `mpwc_newline`. To check whether this option is on, use `__option (mpwc_newline)`. By default, this option is off.

See also "mpwc_newline" on page 142, and "Checking Options" on page 198.

For more information on issues relating to compatibility with MPW in Mac OS programming, see *Targeting Mac OS*.

Relaxed Pointer Type Rules

When you turn on the **Relaxed Pointer Type Rules** option in the C/C++ Language Panel, the compiler treats `char*` and `unsigned char*` as the same type. While prototypes are checked for compatible pointer types, direct pointer assignments will be allowed.

This option is especially useful if you're using code written before the ANSI C standard. Old source code frequently uses these types interchangeably.

This option has no effect on C++. When compiling C++ source code, the compiler differentiates `char*` and `unsigned char*` data types even if the relaxed pointer option is on.

The **Relaxed Pointer Type Rules** option corresponds to the pragma `mpwc_relax`. To check whether this option is on, use `__option (mpwc_relax)`.

See also "mpwc_relax" on page 143, and "Checking Options" on page 198.

Use Unsigned Chars

The C compiler can automatically treat a `char` declarations as `unsigned char`. This behavior is controlled by the **Use Unsigned Chars** setting in the C/C++ Language Panel.

When the Use Unsigned Chars option is on, the C compiler treats a `char` declaration as if it were an `unsigned char` declaration.

NOTE: If you turn this option on, your code may not be compatible with libraries that were compiled with this option turned off.

The **Use Unsigned Chars** option corresponds to the pragma `unsigned_char`. To check whether this option is on, use `__option (unsigned_char)`. By default, this option is off.

See also "unsigned_char" on page 182 and "Checking Options" on page 198.

Using 64-bit Integers

The C compiler lets you define a 64-bit integer with the type specifier `long long`. This behavior is controlled by the `longlong` pragma. There is no item in the C/C++ Language Panel to control this option.

If this option is on, you may declare a `long long` integer. A `long long` can hold values from `-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807`. An `unsigned long long` can hold values from `0` to `18,446,744,073,709,551,615`.

If this option is off, using `long long` causes a syntax error.

In an enumerated type, you can use an enumerator large enough for a `long long`. For more information, see "Enumerated Types" on page 33. However, `long long` bitfields are not supported.

You control the `long long` type with pragma `longlong`.To check whether this option is on, use `__option (longlong).` By default, this pragma is on.

See also "longlong" on page 136 and "Checking Options" on page 198.

Converting Pointers to Types of the Same Size

The C compiler allows the conversion of pointer types to integral data types of the same size in global initializations. Since this type of converstion doesn't conform to the ANSI C standard, it is only

available if the **ANSI Strict** option is off in the **C/C++ Language** settings panel. See "ANSI Strict" on page 37 for more information on this option.

Listing 3.4 Converting a pointer to a same-sized integral type

```
char c;
long arr = (long)&c; // accepted (not ANSI C)
```

Getting Alignment and Type Information at Compile-Time

The C compiler has two built-in functions that return information about a data type's byte alignment and its data type.

The function call `__builtin_align(`*typeID*`)` returns the byte-alignment used for the data type *typeID*.

The function call `__builtin_type(`*typeID*`)` returns an integral value that describes what kind of data type *typeID* is. If *typeID* is an integral oran enumerated type, `__builtin_type(`*typeID*`)` returns 0. If *typeID* is a floating point type, `__builtin_type(`*typeID*`)` returns 1. If *typeID* is any other kind of data type, `__builtin_type(`*typeID*`)` returns 2.

Arrays of Zero Length in Structures

If the **ANSI Strict** option is off in the **C/C++ Language Panel**, the compiler allows arrays of no length as the last item in a structure. Listing 3.5 shows an example. Arrays may be defined with a zero as the index value or with no index value between brackets.

Listing 3.5 Using zero-length arrays

```
struct listOfLongs {
  long listCount;
  long list[0]; // OK if ANSI Strict is off, [] is OK, too.
}
```

4

C++ Compiler

This chapter describes how the CodeWarrior C++ compiler implements the C++ language.

C++ Compiler Overview

This chapter discusses the CodeWarrior C++ compiler as it applies to all CodeWarrior targets. For the most part, the information in this chapter is equally applicable to any operating system or processor.

Other chapters in this manual discuss other features of the compiler that are specific to particular operating systems and processors. For a complete picture, you need to consider all the information relating your target of interest.

In addition, the C compiler is an integral part of the CodeWarrior C++ compiler. As a result, everything about the C compiler applies equally to C++. This discussion of the C++ compiler does not repeat information on the C Compiler. See "C Compiler Overview" on page 29 for information on the C compiler.

This chapter covers all the features of the compiler that exist in support of the C++ language. In addition to describing those features and how to control them, this chapter also has sections on working with advanced features of C++ such as RTTI, exceptions, and templates.

This chapter contains the following sections:

- "CodeWarrior Implementation of C++" on page 54 describes how CodeWarrior C++ implements certain sections of the C++ standard.
- "Unsupported Extensions" on page 61 describes some extensions to the C++ standard that CodeWarrior C++ does not currently support.

- "Controlling the C++ Compiler" on page 62 describes how to change the compiler's behavior by setting options in the C/C++ Language Panel.
- "Working With C++ Exceptions" on page 66 describes how to use the `try` and `catch` statements to perform exception handling.
- "Working With RTTI" on page 67 describes how to use run-time type information support in your code.
- "Working With Templates" on page 70 describes the best way set up the files that define and declare your templates. It also documents an addition to the C++ standard which lets you explicitly instantiate templates.

For information on using Embedded C++ (EC++) and for strategies on developing smaller C++ programs, see "C++ and Embedded Systems Overview" on page 75.

Many topics contain references to ARM, which is *The Annotated C++ Reference Manual* (Addison-Wesley) by Ellis and Stroustrup. These references show you where to look for more information on the information discussed in that topic.

CodeWarrior Implementation of C++

This section describes how CodeWarrior C++ implements certain parts of the C++ standard, as described in *The Annotated C++ Reference Manual* (Addison-Wesley) by Ellis and Stroustrup. The topics discussed in this section are:

- Implicit Return Statement for main()
- Keyword Ordering
- Additional Keywords
- Conversions in the Conditional Operator
- Default Arguments in Member Functions
- Local Class Declarations with Inline Functions
- Copying and Constructing Class Objects
- Checking for Resources To Initialize Static Data

- Calling an Inherited Member Function

Implicit Return Statement for main()

The compiler adds a

```
return 0;
```

statment in C++ a program's `main()` function if the function returns an `int` result and doesn't end with a user `return` statement.

Examples:

```
int main() { } // equivalent to:
               //   int main() { return 0; }
main() { }     // equivalent to:
               //   int main() { return 0; }
```

The compiler will also enforce an external `int main()` function if ANSI Strict is on in the C/C++ Language Panel.

Keyword Ordering

(ARM §7.1.2, §11.4) If you use either the `virtual` or the `friend` keyword in a declaration, it must be the first word in the declaration. For example:

Listing 4.1 Using the virtual or friend keywords

```
class foo {
  virtual int f0();  // OK
  int virtual f1();  // ERROR
  friend int f2();   // OK
  int friend f3();   // ERROR
}
```

Additional Keywords

(ARM §2.4, ANSI §2.8) In addition to reserving the symbols in §2.4 of the ARM as keywords, CodeWarrior C++ reserves these symbols from §2.8 of the ANSI Draft C++ Standard as keywords:

`bool`	`const_cast`	`dynamic_cast`
`explicit`	`false`	`mutable`
`namespace`	`reinterpret_cast`	`static_cast`
`true`	`typeid`	`using`

Conversions in the Conditional Operator

(ARM §5.16) The compiler does not apply reference conversions to the second and third expressions of the conditional operator. In other words, unless the second and third expressions are numeric types, they must be the same type.

Listing 4.2 A conversion in a conditional operator

```
class base { };
class derived : public base { };

static void foo(derived i)
{
  base      &a = i;
  derived   &b = i, c;
c = (sizeof(0) ? a:b); // ERROR: b is not converted to (base &)
c = (sizeof(0) ? a:(base &)b)    // OK, typecast
}
```

Default Arguments in Member Functions

(ARM, §8.2.6) The compiler does not bind default arguments in a member function at the end of the class declaration. Before the default argument appears, you must declare any value that you use in the default argument expression. For example:

Listing 4.3 Using default arguments in member functions

```
class foo {
  enum A { AA };
  int f(A a = AA); // OK
  int f(B b = BB); // ERROR: BB is not declared yet
  enum B { BB };
};
```

Local Class Declarations with Inline Functions

(ARM, §9.8) If you're declaring a class within a function, the class's inline functions cannot access the outer function's local types or variables. In other words, the compiler inserts the class's inline functions on global scope level. For example:

Listing 4.4 Using local class declarations with inline functions

```
int x;

void foo()
{
  static int s;

  class local {
    int f1() { return s; } // ERROR: cannot access 's'

    int f2() { return local::f1(); } // ERROR: cannot access local
    int f3() { return x; } // OK
  };
}
```

Copying and Constructing Class Objects

(ARM, §12.1, §12.8) The compiler does not generate a copy constructor or a default `operator=` for a simple class. A simple class is a class that:

- Is a base class or is derived only from simple classes

- Has no class members or has only simple class members
- Has no virtual member functions
- Has no virtual base classes
- Has no constructor or destructor

Listing 4.5 Constructors

```
class Simple { int f; };

void simpleFunc (Simple s1)
{
  Simple s2 = Simple(s1); // ERROR: An explicit copy constructor
                          // call. The compiler generates no
                          // default copy constructor.

  Simple s3 = s1;  // OK: The compiler performs a bitwise copy
}
```

The compiler does not guarantee that generated assignment or copy constructors will assign or initialize objects representing virtual base classes only once.

Checking for Resources To Initialize Static Data

Sometimes you create static C++ objects that require certain resources, such as a floating-point unit (FPU). You can check for these resources by creating a function called `__PreInit__()` which the compiler calls before it initializes static data. You cannot check for these resources in your `main()` routine, because the compiler initializes static data before it calls `main()`.

You must declare the `__PreInit__()` function like this:

```
extern "C" void __PreInit__(void);
```

NOTE: This function is not supported when generating code for PowerPC.

For example, this stub checks for a floating-point unit. In this case you would also have to define the functions `HasFPU()` and `DisplayNoFPU()` yourself.

Listing 4.6 Checking for an FPU before initializing static data

```
#include <Types.h>
#include <stdlib.h>

extern "C" void __PreInit__(void);

void __PreInit__(void)
{
  if(!HasFPU())   {
    DisplayNoFPU();  // Display "No FPU" Alert
    abort();         // Abort program exection
  }
}
```

Calling an Inherited Member Function

(ARM, §10.2) If you want to call an inherited virtual member function, rather than the local override of that function, you can do so in two ways. The first way is the recommended method for referring to member functions defined in a base class, or any other parent class. The second way, while more convenient, is not recommended if you intend to use your source code with other compilers.

The standard way to call inherited member functions

The first way is supported by the ANSI/ISO C++ Standard and simply qualifies the member function with its base class.

Assume you have two classes, `MyBaseClass` and `MySubClass`, each implementing a function named `MyFunc()`.

From within a function of `MySubClass`, you can call the base class version of `MyFunc()` this way:

```
MyBaseClass::MyFunc();
```

However, if you change the class hierarchy, this code may break. Assume you introduce an intermediate class, and your hierarchy is now `MyBaseClass`, `MyMiddleClass`, and `MySubClass`. Each has a version of `MyFunc()`. The code above still calls the *original* version of `MyFunc()` in the `MyBaseClass`, bypassing the additional behavior you implemented in `MyMiddleClass`. In all likelihood, this is not what you intend, and this kind of subtlety in the code can lead to unexpected results and bugs that are very hard to locate.

Using inherited keyword to call inherited member functions

NOTE: The `inherited` keyword is not supported by the ANSI/ISO C++ standard and is only implemented for single inheritance with CodeWarrior C++.

You may call the inherited version of `MyFunc()` this way:

```
inherited::MyFunc();
```

With the `inherited` keyword, the compiler identifies the base class at compile time. This line of code would call the immediate base class in both cases: where the base class is `MyBaseClass`, and where the immediate base class is `MyMiddleClass`.

If your class hierarchy changes at a later date and your subclass inherits from a different base class, the immediate base class is still called, despite the change in hierarchy.

The syntax is the following:

```
inherited::func-name(param-list);
```

The statement calls the *func-name* in the class's immediate base class. If class has more than one immediate base class (because of multiple inheritance) and the compiler can't decide which *func-name* to call, the compiler generates an error.

This example creates a Q class that draws its objects by adding behavior to the O class.

Listing 4.7 Using `inherited` to call an inherited member function

```
class O { virtual void draw(Point); }
class Q : O { void draw(Point); }

void O::draw (Point p)
{
  Rect r = { p.x-5, p.y-5, p.x+5, p.y+5 };
  FrameOval(r);         // Draw an O.
}


void Q::draw (Point p)
{
  inherited::draw(p);   // Perform behavior of base class
  MoveTo(p.x, p.y);     // Perform added behavior
  Line(5, 5);
}
```

Make sure to insert this directive before using the `inherited` keyword:

```
#pragma def_inherited on
```

For related information on this pragma see "def_inherited" on page 104.

Unsupported Extensions

The C++ compiler does not currently support this extension to the C++ standard as described in The *Annotated C++ Reference Manual* (Addison-Wesley) by Ellis and Stroustrup:

- Overloading methods `operator new[]` and `operator delete[]` to allocate and deallocate the memory for a whole array of objects at once. Instead, overload `operator new()` and `operator delete()`, which are the functions that `operator new[]` and `operator delete[]` call. (ARM, §5.3.3, §5.3.4)

Controlling the C++ Compiler

This section describes how to change the behavior of CodeWarrior C++ by setting some options in the C/C++ Language Panel. For information on this panel and all its options, see "C/C++ Language Panel" on page 15.

This section contains the following:

- Using the C++ Compiler Always
- Controlling ARM Conformance
- Controlling Exception Handling
- Controlling RTTI
- Using the bool Type
- Controlling C++ Extensions

For more information on Direct to SOM, see *Targeting Mac OS*.

Using the C++ Compiler Always

If you turn on the **Activate C++ Compiler** option in the C/C++ Language Panel, the compiler compiles all the C source files in your project as C++ code. If you turn this option off, the CodeWarrior IDE looks at a file name's suffix to determine whether to use the C or C++ compiler. These are the suffixes it looks for:

- If the suffix is .cp, `.cpp`, or `.c++`, the CodeWarrior IDE uses C++
- If the suffix is `.c`, the CodeWarrior IDE uses C.

This option corresponds to the pragma `cplusplus`. To check whether this option is on, use `__option (cplusplus)`. By default, this option is off.

See also "cplusplus" on page 101 and "Checking Options" on page 198.

Controlling ARM Conformance

When the **ARM Conformance** option in the C/C++ Language Panel is on, CodeWarrior C++ generates an error when it encounters certain ANSI C++ features that conflict with the C++ specification in *The Annotated C++ Reference Manual*. Use this option only if you must make sure that your code strictly follows the specification in *The Annotated C++ Reference Manual.*

Turning on this option prevents you from doing the following

- Using protected base classes (ARM, §11.2). For example:

```
class X {};
class Y : protected X {}; // OK in CodeWarrior C++. Error in ARM.
```

- Changing the syntax of the conditional operator to let you use assignment expressions without parentheses in the second and third expressions (K&R, §A7.16). For example:

```
i ? x=y : y=z   // OK in CodeWarrior C++. Error in ARM.
i ? (x=y):(y=z) // OK in ARM and CodeWarrior C++
```

- Declaring variables in the conditions of `if`, `while` and `switch` statements (K&R, §A9.4, §A9.5). For example:

```
while (int i=x+y) { /* ... */ }
       // OK in CodeWarrior C++. Error in ARM.
```

Turning on this option *allows* you to do the following:

- Using variables declared in the condition of a `for` statement after the `for` statement (K&R, §9.5). For example:

```
for(int i=1; i<1000; i++) { /* ... */ }
return i;  // OK in ARM, Error in CodeWarrior C++
```

This option corresponds to the pragma `ARM_conform`. To check whether this option is on, use `__option (ARM_conform)`. By default, this option is off.

See also "ARM_conform" on page 95 and "Checking Options" on page 198.

Controlling Exception Handling

Turn on the **Enable C++ Exceptions** option in the C/C++ Language Panel if you use the ANSI-standard `try` and `catch` statements. Otherwise, turn off this option to generate smaller and faster code.

TIP: If you use PowerPlant for Mac OS programming, make sure this option is turned on. PowerPlant uses C++ exceptions.

For more information on CodeWarrior implements ANSI C++ exception handling mechanism, see "Working With C++ Exceptions" on page 66.

This option corresponds to the pragma `exceptions`. To check whether this option is on, use `__option (exceptions)`. By default, this option is off.

See also "exceptions" on page 116 and "Checking Options" on page 198.

Controlling RTTI

CodeWarrior C++ supports run-time type information (RTTI), including the `dynamic_cast` and `typeid` operators. To use these op-

erators, turn on the **Enable RTTI** option in the C/C++ Language Panel.

For more information on how to use these two operators, see "Working With RTTI" on page 67.

Using the bool Type

Turn on the **Enable bool Support** option if you want to use the standard C++ `bool` type to represent `true` and `false`. Turn this option off if recognizing `bool`, `true`, or `false` as keywords would cause problems in your program.

This option corresponds to the pragma `bool`. To check whether this option is on, use `__option (bool)`. By default, this option is off.

See also "bool" on page 97 and "Checking Options" on page 198.

Controlling C++ Extensions

The C++ compiler has additional extensions that you can activate. There is no item in the C/C++ Language Panel to activate these extensions. You must turn on the pragma `cpp_extensions`.

If this pragma is on, the compiler lets you use these extensions to the ANSI C++ standard:

- Anonymous structs (ARM, §9). For example:

```
#pragma cpp_extensions on
void foo()
{
  union {
    long       hilo;
    struct { short hi, lo; };
      //  annonymous struct
  };
  hi=0x1234;
  lo=0x5678;
```

```
      //  hilo==0x12345678
}
```

- Unqualified pointer to a member function (ARM, §8.1c). For example:

```
#pragma cpp_extensions on
struct Foo { void f(); }
void Foo::f()
{
  void (Foo::*ptmf1)() = &Foo::f;
    // ALWAYS OK

  void (Foo::*ptmf2)() = f;
    // OK, if cpp_exptensions is on.
}
```

To check whether this option is on, use the `__option (cpp_extensions)`. By default, this option is off.

See also "cpp_extensions" on page 101 and "Checking Options" on page 198.

Working With C++ Exceptions

If you turn on the **Enable C++ Exceptions** options in the C/C++ Language Panel, you may use the `try` and `catch` statements to perform exception handling. For more information on activating support for C++ exception handling, see "Controlling Exception Handling" on page 64.

If exceptions are enabled, you can throw exceptions across any code that's compiled by the CodeWarrior C/C++ compiler. You cannot throw exceptions across the following:

- Mac OS Toolbox function calls
- Libraries compiled with exception support turned off
- Libraries compiled with versions of the CodeWarrior C/C++ compiler earlier than CodeWarrior 8

- Libraries compiled with CodeWarrior Pascal or other compilers

If you throw an exception across one of these, the code calls `terminate()` and exits.

If you throw an exception when you're allocating a class object or an array of class objects, the code automatically destructs the partially constructed objects and de-allocates the memory for them.

Working With RTTI

This section describes how to work with run-time type information features of C++ supported in the CodeWarrior C++ compiler. RTTI lets you cast one type of object to be another type, get information about objects, and compare their types at runtime.

The topics in this section are:

- Using the dynamic_cast Operator
- Using the typeid Operator

Using the dynamic_cast Operator

The `dynamic_cast` operator lets you safely convert a pointer of one type to a pointer of another type. Unlike an ordinary cast, `dynamic_cast` returns 0 if the conversion is not possible. An ordinary cast returns an unpredictable value that may crash your program if the conversion is not possible.

This is the syntax for `dynamic_cast` operator:

`dynamic_cast<`*Type*`*>(`*expr*`)`

The *Type* must be either `void` or a class with at least one virtual member function. If the object that *expr* points to (`*`*expr*) is of type *Type* or is derived from type *Type*, this expression converts *expr* to a pointer of type *Type*`*` and returns it. Otherwise, it returns `0`, the null pointer.

For example, take these classes:

```
class Person { virtual void func(void) { ; } };
class Athlete : public Person { /* . . . */ };
class Superman : public Athlete { /* . . . */ };
```

And these pointers:

```
Person *lois = new Person;
Person *arnold = new Athlete;
Person *clark = new Superman;
Athlete *a;
```

This is how `dynamic_cast` would work with each:

```
a = dynamic_cast<Athlete*>(arnold);
  // a is arnold, since arnold is an Athlete.
a = dynamic_cast<Athlete*>(lois);
  // a is 0, since lois is not an Athelete.
a = dynamic_cast<Athlete*>(clark);
  // a is clark, since clark is both a Superman and an Athlete.
```

You can also use the `dynamic_cast` operator with reference types. However, since there is no equivalent to the null pointer for references, `dynamic_cast` throws an exception of type `bad_cast` if it cannot perform the conversion.

NOTE: The `bad_cast` type is defined in the header file exception. Whenever you use `dynamic_cast` with a reference, you must #include exception.

This is an example of using `dynamic_cast` with a reference:

```
#include <exception>
// . . .
Person &superref = *clark;

try {
```

```
  Person &ref = dynamic_cast<Person&>(superref);
}
catch(bad_cast) {
  cout << "oops!" << endl;
}
```

Using the typeid Operator

The `typeid` operator lets you determine the type of an object. Like the `sizeof` operator, it takes two kinds of arguments:

- the name of a class
- an expression that evaluates to an object

NOTE: Whenever you use `typeid` operator, you must #include the typeinfo header file.

The `typeid` operator returns a reference to a `type_info` object that you can compare with the == and `!=` operators. For example, if you have these classes and objects:

```
class Person { /* . . . */ };
class Athlete : public Person { /* . . . */ };

Person *lois = new Person;
Athlete *arnold = new Athlete;
Athlete *louganis = new Athlete;
```

All these expressions are true:

```
#include <typeinfo>
// . . .
if (typeid(Athlete) == typeid(*arnold))
   // arnold is an Athlete, result is true
if (typeid(*arnold) == typeid(*louganis))
   // arnold and louganis are both Athletes, result is true
```

```
if (typeid(*lois) == typeid(*arnold)) // ...
   // lois and arnold are not the same type, result is false
```

You can access the name of a type with the `name()` member function in the `type_info` class. For example, these statements:

```
#include <typeinfo>
// . . .
cout << "Lois is a(n) "
     << typeid(*lois).name() << endl;
cout << "Arnold is a(n) "
     << typeid(*arnold).name() << endl;
```

Print this:

```
Lois is a(n) Person
Arnold is a(n) Athlete
```

Working With Templates

(ARM, §14) This section describes the best way to organize your template declarations and definitions in files. It also documents how to explicitly instantiate templates, using a syntax that is not in the ARM but is part of the ANSI C++ draft standard.

This section includes the following topics:

- Declaring and Defining Templates
- Instantiating a Template

Declaring and Defining Templates

In a header file, declare your class functions and function templates, as shown in Listing 4.8.

Listing 4.8 templ.h: A Template Declaration File

```
template <class T>
class Templ {
  T  member;
public:
  Templ(T x) { member=x; }
  T  Get();
};

template <class T>
T Max(T,T);
```

In a source file, include the header file and define the function templates and the member functions of the class templates. Listing 4.9 shows you an example.

This source file is a template definition file. You'll include this file in any file that uses your templates. You do not need to add the template definition file to your project. Although this is technically a source file, you work with it as if it were a header file.

The template definition file does *not* generate code. The compiler cannot generate code for a template until you specify what values it should substitute for the templates arguments. Specifying these values is called instantiating the template. See "Instantiating a Template" on page 72.

Listing 4.9 templ.cp: A Template Definition File

```
#include "templ.h"

template <class T>
T Templ<T>::Get()
{
  return member;
}

template <class T>
T Max(T x, T y)
```

```
{
  return ((x>y)?x:y);
}
```

WARNING! Do not include the original template declaration file, which ends in `.h`. If you include the template `.h` file in your source file, the compiler will generate an error saying that the function or class is undefined.

Instantiating a Template

The compiler cannot generate code for a template until you:

- declare the template class
- provide a template definition
- specify the data type(s) for the template

For information on the first two requirements, see "Declaring and Defining Templates" on page 70.

Specifying the data type(s) and other arguments for a template is called instantiating the template. CodeWarrior C++ gives you two ways to instantiate a template. You can let the compiler instantiate it automatically when you first use it, or you can explicitly create all the instantiations you'll need.

Automatic instantiation

To instantiate templates automatically, include the template definition file in all the source files that use the template, and just use the template members as you would any other type or function. The compiler automatically generates code for a template instantiation whenever it sees a new one. Listing 4.10 shows how to automatically instantiate the templates in Listing 4.8 and Listing 4.9, class `Templ` and class `Max`.

Listing 4.10 myprog.cp: A Source File that Uses Templates

```
#include <iostreams.h>
#include "templ.cp" // includes templ.h as well

void main(void) {
  Templ<long> a = 1, b = 2;
    // The compiler instantiates Templ<long> here.
  cout << Max(a.Get(), b.Get());
    // The compiler instantiates Max<long>() here.
}
```

If you use automatic instantiation, the compiler may take longer to compile your program because it has to determine on its own which instantiations you'll need. Also, the object code for the template instantiations will be scattered throughout your program.

Explicit instantiation

To instantiate templates explicitly, include the template definition file in a source file, and write a template instantiation statement for every instantiation. The syntax for a class template instantiation is

```
template class class-name<templ-specs>;
```

The syntax for a function template instantiation is

```
template return-type func-name<templ-specs>(arg-specs)
```

Listing 4.11 shows how to explicitly instantiate the templates in Listing 4.8 and Listing 4.9.

Listing 4.11 myinst.cp: Explicitly Instantiating Templates

```
#include "templ.cp"

template class Templ<long>; // class instantiation
template long Max<long>(long,long); // function instantiation
```

When you're explicitly instantiating a function, you do not need to include in *templ-specs* any arguments that the compiler can deduce from *arg-specs*. For example, in Listing 4.11 you can instantiate `Max<long>()` like this:

```
template long Max<>(long, long);
  // The compiler can tell from the arguments
  // that you're instantiating Max<long>()
```

If you use explicit instantiation, the compiler compiles your program more quickly. Because the instantiations can be in one file with no other code, you can even choose to put them in a separate library.

NOTE: Explicit instantiation is not in the ARM but is part of the ANSI C++ draft standard.

5

C++ and Embedded Systems

This chapter describes how to develop effective software for embedded systems using CodeWarrior C++. It also has topics that all C++ programmers may find useful for developing smaller programs.

C++ and Embedded Systems Overview

This chapter covers the following items of concern to embedded systems programmers.

- Activating EC++
- Differences Between ANSI/ISO C++ and EC++
- Meeting EC++ Specifications With CodeWarrior
- Strategies for Smaller Code Size in C++

NOTE: This chapter discusses some strategies for program design for embedded systems and is not meant to be a definitive solution.

Currently, CodeWarrior C++ may be used for EC++-compatible embedded systems development, but does not include some libraries mentioned in the EC++ proposal.

Activating EC++

To compile EC++ source code, make sure the **EC++ Compatibility Mode** option is on in the **C/C++ Language** settings panel.

To test for EC++ compatibility mode at compile time use the __embedded_cplusplus predefined symbol. For more information, see "Predefined Symbols" on page 193.

Differences Between ANSI/ISO C++ and EC++

Several features of ANSI/ISO C++ (ANSI C++) are not present in EC++. Among the features not supported in EC++ are:

- Templates
- Libraries
- File Operations
- Localization
- Exception Handling
- Other Language Features

Templates

ANSI C++ supports templates. The EC++ proposal does not include template support for class or functions.

Libraries

The classes `<string>`, `<complex>`, `<ios>`, `<streambuf>`, `<istream>` and `<ostream>` are supported in Embedded C++ specifications. However only in a non-template form. All other ANSI C++ libraries including the STL-type algorithm libraries are not supported

File Operations

There are no file operations specified in the EC++ proposed standard except for simple console input and output file types.

Localization

There are no localization libraries in the EC++ proposed standard because of the excessive memory requirements.

Exception Handling

Exception handling is not supported in the EC++ proposed standard.

Other Language Features

The following language features are not supported. Some other minor features are also unsupported but not listed.

- mutable specified
- RTTI
- namespace
- multiple inheritance
- virtual inheritance

Meeting EC++ Specifications With CodeWarrior

This section describes how to be compliant with the proposed Embedded C++ (EC++). These topics discuss different facets of designing software to meet the EC++ standard:

- Language Related Issues
- Library Related Issues

Language Related Issues

To make sure your source code is compatible with both the ANSI/ISO C++ and EC++ standards, follow these guidelines:

- do not use RTTI (Run-Time Type Identification).
- do not use exception handling, namespaces, or other unsupported features.
- do not use multiple or virtual inheritance

Some of these C++ features, such as RTTI and exceptions, can be turned off in the compiler settings for your code. These options are

in the C++ Language settings panel, described in "Setting C Compiler Options Overview" on page 15.

Library Related Issues

Do not refer to routines, data structures, and classes in the Metrowerks Standard Libraries for C++.

Metrowerks will explore alternative class libraries that are more suitable for use with EC++-compliant applications and may make them available in a future release.

Strategies for Smaller Code Size in C++

When using C++ where program size may be critical, there are certain strategies that a programmer can follow to ensure optimal code size.

NOTE: In all strategies, reducing the size of object may affect the performance of that code.

Some of these strategies are used by the proposed Embedded C++ (EC++) uses some of these strategies as part of its specification. Other strategies apply to C++ programming in general. All of these strategies may be used by any C++ program, whether the program follows the EC++ standard or not.

In CodeWarrior, you can group these into compiler-related, language-related, and library-related steps.

Compiler-related strategies

Compiler-related strategies rely on using compiler features to reduce the size of object code.

- Size Optimizations—use the compiler size optimization settings

- Inlining—how to control and limit the effectiveness of the `inline` directive

Language-related strategies

Language-related strategies limit or avoid the use of ANSI/ISO C++ features. Although these features may make software design and maintenance easier, they often do so at the cost of affecting code size

- Virtual Functions—not using virtual functions reduces the size of code
- Runtime Type Identification—the compiler won't generate extra data if a program doesn't use Runtime Type Identification (RTTI)
- Exception Handling—while CodeWarrior C++ provides zero-overhead exception handling to provide optimum execution speed, it still generates extra object code for exception support
- Operator New—be careful not to throw an exception within the `new` operator
- Multiple Inheritance—the compiler won't generate extra data if the use of multiple inheritance isn't used

Library-related strategies

- Stream-Based Classes—these classes in the Metrowerks Standard Libraries comprise a lot of object code
- Alternative Class Libraries—non-standard class libraries may provide a subset of the standard library's functionality with less overhead

Size Optimizations

Metrowerks compilers include optimization settings for size or speed, and various levels of optimization. Choose size as your desired outcome, and the level of optimization you wish to apply.

You control optimization settings as an option in the settings for your target. The option is in the Processor settings panel.

When debugging, compile your code without any optimizations. Some optimizations disrupt the relationship between source and object code required by the debugger. Optimize your code after you have finished debugging.

See also "Setting C Compiler Options Overview" on page 15.

Inlining

With CodeWarrior you can turn inlining off, allow normal inlining, auto-inline, or set the maximum depth of inlining.

Inlining may reduce or increase code size. There is no definite answer for this question. Inlining small functions can make a program smaller. In particular if you have a class library with a lot of getter/setter member functions, the code size can be quite a bit smaller with inlining on.

However, MSL C++ defines many functions as inline, and this is not good if you want minimal code size. For optimal code size when using MSL C++, turn inlining off when you build the library. If you are not using MSL C++, normal inlining and a common sense use of the keyword "inline" may improve your code size.

In CodeWarrior you control inlining as a language option in the target settings. The option is in the C/C++ Language settings panel.

When debugging your code, turn inlining off to maintain a clear correspondence between source and object code. After debugging, set the inlining level that has the best effect on your object code.

See also "Inlining" on page 43.

Virtual Functions

For optimal code size virtual functions should not be used except when necessary. A virtual function is never dead-stripped, even if it's never called.

Runtime Type Identification

If code size is an issue, do not use RTTI. RTTI generates a data table for every class, so turning this off will make the data section smaller.

The proposal for Embedded C++ does not allow runtime type identification. Turn RTTI off as a C++ language option in the target settings. The option is in the C/C++ Language settings panel.

See also "Controlling RTTI" on page 64.

Exception Handling

Be selective when using C++ exception handling routines, or not use exceptions at all. CodeWarrior has a zero runtime overhead error handling mechanism. However, using exceptions still adds some code size. The exception tables (data) can get pretty big when using exception handling.

The proposal for Embedded C++ does not allow exception handling. Turn exception handling off as a C++ language option in the target settings. The option is in the C/C++ Language settings panel.

NOTE: The proposed ANSI/ISO standard libraries and the use of the "new" operator will require exception handling. See "Operator New" on page 81.

Operator New

The C++ `new` operator might throw an exception or not depending on how the runtime library implements the `new` operator. To have it throw exceptions set `__throws_bad_alloc` to `1`, to have it not set `__throws_bad_alloc` to `0` in the prefix file for your target and rebuild your library.

Please read your release notes or *Targeting* manual for more information.

Multiple Inheritance

The code and data overhead required to implement multiple inheritance is fairly modest.

The proposal for Embedded C++ does not allow multiple inheritance.

Virtual Inheritance

For optimal code size, do not use virtual inheritance. Virtual base classes are often complex and will add a lot of code to the constructor and destructor functions.

The proposal for Embedded C++ does not allow virtual inheritance.

Stream-Based Classes

The Metrowerks Standard Library (MSL) C++ stream-based classes will initialize several instances of direct and indirect objects. When code size is critical, do not use any stream-based classes. The stream-based classes include standard input (`cin`), standard output (`cout`), and standard error (`cerr`). There are wide-character equivalents for the normal input and output routines as well. Unless there is a great need for these classes, use standard C input and output functions instead.

In addition to the standard C++ stream classes, there are string streams for in-core formatting that will also evoke a heavy overhead and should be avoided. If size is critical, use C's `sprintf` or `sscanf` functions instead.

The proposal for Embedded C++ does not allow for templatized classes or functions. MSL is compliant with the ANSI/ISO proposed standards that are based on templates.

Alternative Class Libraries

Metrowerks Standard Library (MSL) C++ is based on the ANSI/ISO proposed C++ standard. This C++ Standard is implemented using templates, which have a large initial overhead for specialization.

To avoid this overhead, you may want to devise your own vector, string, or other utility classes that you use commonly. In addition, there are other class libraries available, such as the NIH's (National Institute of Health) Class Library. These may be more suitable for your work.

If you do use an alternative library, keep in mind possible problems with virtual inheritance, RTTI, or other causes of larger code size that we described above.

6

Pragmas and Symbols

This chapter describes the pragmas and predefined symbols available with the Metrowerks C/C++ compiler.

Pragmas and Symbols Overview

You set compiler options for an entire project by changing the settings in the C/C++ Language Panel. You can also control how the compiler compiles your code from within your code, using pragmas.

Most of the pragmas correspond to options in the C/C++ Language Panel, or other settings panels such as the 68K Processor panel, the PowerPC processor panel, and so forth.

Typically, you use the settings panels to set the options for most of your code and use pragmas to change the options for special cases. For example, with the C/C++ Language Panel, you can turn off a time-consuming optimization and, with a pragma, turn it on only for the code it helps most.

TIP: If you use Metrowerks command-line tools, such as those for MPW or Be OS, see the Command-Line Tools manual for information on how to duplicate the effect of #pragma statements using command-line tool options.

The sections in this chapter are:

- Pragma Syntax—explains pragma syntax
- Pragma Scope—the range of a pragma

- Pragmas—lists each pragma
- Predefined Symbols—lists ANSI and CodeWarrior symbols
- Checking Options—explains how to determine the current setting of most pragmas and options

Pragmas

This section describes how to use pragmas and lists and explains each pragma.

These topics describe how to use pragmas:

- Pragma Syntax
- Pragma Scope

The pragmas are:

a6frames	align	align_array_members
always_inline	ANSI_strict	arg_dep_lookup
ARM_conform	auto_inline	bool
check_header_flags	code_seg	code68020
code68881	cplusplus	cpp_extensions
enumsalwaysint	d0_pointers	data_seg
def_inherited	defer_codegen	define_section
direct_destruction	direct_to_som	disable_registers
dont_inline	dont_reuse_strings	dollar_identifiers
ecplusplus	EIPC_EIPSW	enumsalwaysint
exceptions	export	extended_errorcheck
far_data	far_code, near_code, smart_code	far_vtables
faster_pch_gen	force_active	fourbyteints
fp_contract	fp_pilot_traps	function

global_optimizer, optimization_level
IEEEdoubles
ignore_oldstyle
import
init_seg
inline_depth
inline_intrinsics
internal
interrupt
k63d
k63d_calls
lib_export
longlong
longlong_enums
macsbug, oldstyle_symbols
mark
microsoft_exceptions
microsoft_RTTI
mmx
mmx_call
mpwc
mpwc_newline
mpwc_relax
no_register_coloring
once
only_std_keywords
opt_common_subs
opt_dead_assignments
opt_dead_code
opt_lifetimes
opt_loop_invariants
opt_propagation
opt_strength_reduction
opt_unroll_loops
opt_vectorize_loops
optimize_for_size
oldstyle_symbols
optimization_level
pack
parameter
peephole
pointers_in_A0, pointers_in_D0
pool_strings
pop, push
precompile_target
profile
readonly_strings
register_coloring
require_prototypes
RTTI
scheduling
section
segment
side_effects
simple_prepdump
SOMCallOptimization
SOMCallStyle
SOMCheckEnvironment
SOMClassVersion
SOMMetaClass
SOMReleaseOrder
stack_cleanup
static_inlines
suppress_init_code
sym
syspath_once
toc_data
traceback
trigraphs

unsigned_char	unused	use_fp_instructions
use_frame	use_mask_registers	warning_errors
warn_emptydecl	warn_extracomma	warn_hidevirtual
warn_illpragma	warn_implicitconv	warn_notinlined
warn_possunwant	warn_structclass	warn_unusedarg
warn_unusedvar	warning	
profile	readonly_strings	register_coloring
require_prototypes	RTTI	scheduling
section	segment	side_effects
simple_prepdump	SOMCallOptimization	SOMCallStyle
SOMCheckEnvironment	SOMClassVersion	SOMMetaClass

Pragma Syntax

Most pragmas have this syntax:

```
#pragma option-name on | off | reset
```

Generally, use `on` or `off` to change the option's setting, and then use `reset` to restore the option's original setting, as shown below:

```
#pragma profile off
  // If the option Generate Profiler Calls is on,
  // turns it off for these functions.

#include <smallfuncs.h>

#pragma profile reset
  // If the option Generate Profiler Calls was originally on,
  // turns it back on. Otherwise, the option remains off
```

Suppose that you use `#pragma profile on` instead of `#pragma profile reset`. If you later turn off **Generate Profiler Calls** from

the Preference dialog, that pragma turns it on. Using `reset` ensures that you don't inadvertently change the settings in the Project Settings dialog.

Pragma Scope

In general, the scope of a pragma setting is limited to a single file.

As discussed in Pragma Syntax, you should use `on`, or `off` after the pragma's name to change a pragma's setting to the condition you want. After you have set the pragma to the desired state, all code after that point is compiled with that setting until either:

- change the setting with `on`, `off`, or (preferred) `reset`
- you reach the end of the file

At the beginning of each file, the compiler reverts to the project or default settings.

Pragma settings are not stored in a precompiled header file. In other words, you can modify compiler settings inside a precompiled header, but the change affects the code within that file only. The settings that were active at the end of the precompiled header file are lost and have to be set up again.

The settings for an item that is declared in a precompiled header file (such as data or a function) are saved and restored when the precompiled header file is included.

For example, this code says that the variable `xxx` is a far variable.

```
// in file pch.h

#pragma far_data on
extern int xxx;
```

Now, assume a file includes this as a precompiled header.

```
// in file test.c
#pragma far_data off // far data is off
```

```
#include "pch.pch" // this file set far_data on

// far_data is still 'off' but xxx is still a far variable
```

The pragma setting still works within the header file, even though the source file including the header has a different setting.

a6frames

Description Controls the generation of stack frames based on the A6 register.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma a6frames on | off | reset
```

Remarks This pragma applies to Mac OS on 68K programming only.

If this pragma is on, the compiler generates `A6` stack frames which let debuggers trace through the call stack and find each routine. Many debuggers, including the Metrowerks debugger and Jasik's The Debugger, require these frames. If this pragma is off, the compiler does not generate these frames, so the generated code is smaller and faster.

This is the code that the compiler generates for each function, if this pragma is on:

```
LINK #nn,A6
UNLK A6
```

This pragma corresponds to **Generate A6 Stack Frames** option in the 68K Linker settings panel. To check whether this option is on, use `__option (a6frames)`, described in "Checking Options" on page 198.

align

Description Specifies how to align data.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma options align=` *alignment*

Remarks This pragma applies to Mac OS programming only.

This pragma specifies how to align structs and classes, where *alignment* can be one of the following values:

If alignment is	The compiler …
`mac68k`	Aligns every field on a 2-byte boundaries, unless a field is only 1-byte long. This is the standard alignment for 68K Macintosh computers.
`mac68k4byte`	Aligns every field on 4-byte boundaries.
`power`	Align every field on its natural boundary. This is the standard alignment for Power Macintosh computers. For example, it aligns a character on a 1-byte boundary and a 16-bit integer on a 2-byte boundary. The compiler applies this alignment recursively to structured data and arrays containing structured data. So, for example, it aligns an array of structured types containing an 4-byte floating point member on an 4-byte boundary.
`native`	Aligns every field using the standard alignment. It is equivalent to using `mac68k` for 68K Macintosh computers and `power` for Power Macintosh computers.

If alignment is	The compiler …
`packed`	Aligns every field on a 1-byte boundary. It is not available in any settings panel. This alignment will cause your code to crash or run slowly on many platforms. ***Use it with caution.***
`reset`	Resets the option to the value in the previous `#pragma options align` statement, if there is one, or to the value in the 68K or PPC Processor settings panel.

Note there is a space between `options` and `align`.

This pragma corresponds to the **Struct Alignment** option in the 68K Processor settings panel.

align_array_members

Description Controls the alignment of arrays within `struct` and `class` data.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma align_array_members on | off | reset
```

Remarks This pragma applies to Mac OS programming only.

This option lets you choose how to align an array in a struct or class. If this option is on, the compiler aligns all array fields larger than a byte according to the setting of the **Struct Alignment** option. If this option is off, the compiler doesn't align array fields.

Listing 6.1 Choosing how to align arrays

```
#pragma align_array_members off
struct X1 {
        char c;         // offset==0
```

```
        char arr[4];    // offset==1 (char aligned)
};

#pragma align_array_members on
#pragma align mac68k
struct X2 {
        char c;         // offset==0
        char arr[4];    // offset==2 (2-byte align)
};

#pragma align_array_members on
#pragma align mac68k4byte
struct X3 {
        char c;         // offset==0
        char arr[4];    // offset==4 (4-byte align)
};
```

To check whether this option is on, use `__option (align_array_members)`, described in "Checking Options" on page 198. By default, this option is off.

always_inline

Description Controls the use of inlined functions.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma always_inline on | off | reset
```

Remarks When this option is on, the compiler attempts to inline every function declared with the `inline` keyword.

This pragma doesn't correspond to any settings panel option. To check whether this option is on, use `__option (always_inline)`, described in "Checking Options" on page 198.

ANSI_strict

Description Controls the use of non-standard language features.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma ANSI_strict on | off | reset
```

Remarks The common ANSI extensions are the following. If you turn on the pragma `ANSI_strict`, the compiler generates an error if it encounters any of these extensions.

- C++-style comments. For example:

```
a = b;    // This is a C++-style comment
```

- Unnamed arguments in function definitions. For example:

```
void f(int ) {}  /* OK, if ANSI Strict is off */
void f(int i) {} /* ALWAYS OK                 */
```

- A # token not followed by an argument in a macro definition. For example:

```
#define add1(x) #x #1
      /* OK, if ANSI_strict is off,
         but probably not what you wanted:
         add1(abc) creates "abc"#1            */

#define add2(x) #x "2"
      /* ALWAYS OK: add2(abc) creates "abc2"      */
```

- An identifier after `#endif`. For example:

```
#ifdef __MWERKS__
  /* . . . */
#endif __MWERKS__       /* OK, if ANSI_strict is off */
```

```
#ifdef __MWERKS__
  /* . . . */
#endif /*__MWERKS__*/   /* ALWAYS OK  */
```

This pragma corresponds to the **ANSI Strict** option in the C/C++ Language Panel. To check whether this option is on, use `__option(ANSI_strict)`, described in "Checking Options" on page 198.

arg_dep_lookup

Description Controls C++ argument-dependent name lookup.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma arg_dep_lookup on | off | reset
```

Remarks When this option is on, the C++ compiler uses argument-dependent name lookup. By default this option is on.

This pragma doesn't correspond to any settings panel option. To check whether this option is on, use `__option(arg_dep_lookup)`, described in "Checking Options" on page 198.

ARM_conform

Description Controls the use of non-ARM language features.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma ARM_conform on | off | reset
```

Remarks When pragma `ARM_conform` is on, the compiler generates an error when it encounters certain ANSI C++ features that conflict with the

C++ specification in *The Annotated C++ Reference Manual*. Use this option only if you must make sure that your code strictly follows the specification in *The Annotated C++ Reference Manual.*

Turning on this pragma prevents you from doing the following

- Using protected base classes. For example:

```
class X {};
class Y : protected X {}; // OK if ARM_conform is off.
```

- Changing the syntax of the conditional operator to let you use assignment expressions without parentheses in the second and third expressions. For example:

```
i ? x=y : y=z  // OK if ARM_conform is off.
i ? (x=y):(y=z)   // ALWAYS OK
```

- Declaring variables in the conditions of `if`, `while` and `switch` statements. For example:

```
while (int i=x+y) { . . . }  // OK if ARM_conform is off.
```

Turning on this option *allows* you to do the following:

- Using variables declared in the condition of an `if` statement after the `if` statement. For example:

```
for(int i=1; i<1000; i++) { /* . . . */ }
return i;  // OK if ARM_conform is on.
```

This pragma corresponds to the **ARM Conformance** option in the C/C++ Language Panel. To check whether this option is on, use `__option (ARM_conform)`, described in "Checking Options" on page 198.

auto_inline

Description Controls the selection of which functions to inline.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma auto_inline on | off | reset
```

Remarks If this pragma is on, the compiler, automatically picks functions to inline for you

Note that if either the **Don't Inline** option ("Inlining" on page 43) or the `dont_inline` pragma ("dont_inline" on page 113) is on, the compiler ignores the setting of the `auto_inline` pragma and doesn't inline any functions.

This pragma corresponds to the **Auto-Inline** option of the **Inlining** menu the C/C++ Language Panel. To check whether this option is on, use `__option (auto_inline)`, described in "Checking Options" on page 198.

bool

Description Controls if `bool`, `true`, and `false` are treated as keywords or not.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma bool on | off | reset
```

Remarks When this pragma is on, you can use the standard C++ `bool` type to represent `true` and `false`. Turn this pragma off if recognizing `bool`, `true`, or `false` as keywords would cause problems in your program.

This pragma corresponds to the **Enable bool Support** option in the C/C++ Language Panel, described in "Using the bool Type" on page 65. To check whether this option is on, use `__option(bool)`, described in "Checking Options" on page 198. By default, this option is off.

check_header_flags

Description Sets if checking should be done to ensure that a precompiled header's data matches a project's target settings.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma check_header_flags on | off | reset
```

Remarks This pragma affects precompiled headers only.

When this pragma is `on`, the compiler makes sure that the precompiled header's preferences for `double` size (8-byte or 12-byte), `int` size (2-byte or 4-byte) and floating point math correspond to the build target's settings. If they do not match, the compiler generates an error.

If your precompiled header file has settings that are independent from those in the project, turn this pragma off. If your precompiled header depends on these settings, turn this pragma on.

This pragma does not correspond to any option in the C/C++ Language Panel. To check whether this option is on, use `__option (check_header_flags)`, described in "Checking Options" on page 198. By default, this pragma is off.

code_seg

Description Specifies the segment into which code is placed.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma code_seg(`*name*`)`

Remarks This pragma designates the segment into which compiled code is placed. The *name* is a string specifying the name of the code segment. For example, the pragma

```
#pragma code_seg(".code")
```

places all subsequent code into a segment named `.code`.

code68020

Description Controls object code generation for Motorola 680x0 (and higher) processors.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma code68020 on | off | reset`

Remarks This pragma applies to 68K programming only.

When this option is on, the compiler generates code that's optimized for the MC68020. The code runs on a Power Macintosh or a Macintosh with a MC68020 or MC68040. The code does crash on a Macintosh with a MC68000. When this option is off, the compiler generates code that will run on any Macintosh.

WARNING! Do not change this option's setting within a function definition.

Before your program runs code optimized for the MC68020, use the `gestalt()` function to make sure the chip is available. For more information on `gestalt()`, see Chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.

In the Mac OS compiler, this option is off by default.

This pragma corresponds to the **68020 Codegen** option in the 68K Processor settings panel. To check whether this option is on, use `__option (code68020)`, described in "Checking Options" on page 198.

code68881

Description Controls object code generation for Motorola 68881 (and higher) math coprocessors.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma code68881 on | off | reset
```

Remarks This pragma applies to 68K programming only.

When this option is on, the compiler generates code that's optimized for the MC68881 floating-point unit (FPU). This code runs on a Macintosh with an MC68881 FPU, MC68882 FPU, or a MC68040 processor. (The MC68040 has a MC68881 FPU built in.) The code does not run on a Power Macintosh, a Macintosh with an MC68LC040, or a Macintosh with any other processor and no FPU. When this option is off, the compiler generates code that will run on any Macintosh.

WARNING! If you use the `code68881` pragma to turn this option on, place it at the beginning of your file, before you include any files and declare any variables and functions.

Before your program runs code optimized for the MC68881, use the `gestalt()` function to make sure an FPU is available. For more information on `gestalt()`, see Chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.

This pragma corresponds to the **68881 Codegen** option in the 68K Processor settings panel. To check whether this option is on, use `__option (code68881)`, described in "Checking Options" on page 198.

cplusplus

Description Specifies if subsequent source code should be translated as C or C++ source code.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma cplusplus on | off | reset
```

Remarks When this pragma is on, the compiler compiles the code that follows as C++ code. When this option is off, the compiler uses the suffix of the filename to determine how to compile it. If a file's name ends in `.cp`, `.cpp`, or `.c++`, the compiler automatically compiles it as C++ code. If a file's name ends in `.c`, the compiler automatically compiles it as C code. You need to use this pragma only if a file contains a mixture of C and C++ code.

This pragma corresponds to the **Activate C++ Compiler** option in the C/C++ Language Panel. To check whether this option is on, use `__option (cplusplus)`, described in "Checking Options" on page 198.

cpp_extensions

Description Controls language extensions to ANSI/ISO C++.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma cpp_extensions on | off | reset`

Remarks If this option is on, it enables these extensions to the ANSI C++ standard:

- Anonymous structs. For example:

```
#pragma cpp_extensions on
void foo()
{
  union {
    long  hilo;
    struct { short hi, lo; }; //  annonymous struct
  };
  hi=0x1234;
  lo=0x5678;  //  hilo==0x12345678
}
```

- Unqualified pointer to a member function. For example:

```
#pragma cpp_extensions on
struct Foo { void f(); }
void Foo::f()
{
  void (Foo::*ptmf1)() = &Foo::f; // ALWAYS OK

  void (Foo::*ptmf2)() = f; // OK, if cpp_exptensions is on.
}
```

This pragma does not correspond to any option in the C/C++ Language Panel. To check whether this option is on, use the `__option (cpp_extensions)`, described in "Checking Options" on page 198. By default, this option is off.

d0_pointers

Description Specifies which register should be used to hold function result pointers.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma d0_pointers`

Remarks This pragma applies to 68K programming only.

This pragma lets you choose between two calling conventions: the convention for MPW and Macintosh Toolbox routines and the convention for Metrowerks C and C++ routines. In the MPW and Macintosh Toolbox calling convention, functions return pointers in the register `D0`. In the Metrowerks C and C++ convention, functions return pointers in the register `A0`.

When you declare functions from the Macintosh Toolbox or a library compiled with MPW, turn on the `d0_pointers` pragma. After you declare those functions, turn off the pragma to start declaring or defining Metrowerks C and C++ functions.

In Listing 6.2, the Toolbox functions in `Sound.h` return pointers in `D0` and the user-defined functions in `Myheader.h` use `A0`.

Listing 6.2 Using #pragma pointers_in_A0 and #pragma pointers_in_D0

```
#pragma d0_pointers on     // set for Toolbox calls
#include <Sound.h>
#pragma d0_pointers reset // set for my routines
#include "Myheader.h"
```

The pragmas `pointers_in_A0` and `pointers_in_D0` have much the same meaning as `d0_pointers` and are available for background compatibility. The pragma `pointers_in_A0` corresponds to `#pragma d0_pointers off` and the pragma `pointers_in_D0` corresponds to `#pragma d0_pointers on`. The pragma `d0_pointers` is recommended for new code since it supports the `reset` argument. For more information, see "pointers_in_A0, pointers_in_D0" on page 155.

This pragma does not correspond to any option in the 68K Processor settings panel. To check whether this option is on, use the `__option (d0_pointers)`, described in "Checking Options" on page 198.

data_seg

Description Ignored, but included for compatibility with Microsoft compilers.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma data_seg(name)
```

Remarks Ignored. Included for compatibility with Microsoft. It designates the segment into which initialized is placed. The *name* is a string specifying the name of the data segment. For example, the pragma

```
data_seg(".data")
```

places all subsequent data into a segment named `.data`.

def_inherited

Description Controls the use of the `inherited` keyword.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma def_inherited on | off | reset
```

Remarks This option allows the use of the `inherited` keyword in C++ programming. The default is off.

NOTE: The `inherited` keyword is not supported by the ANSI/ISO C++ standard and is only implemented for single inheritance with CodeWarrior C++.

To check whether this option is on, use the `__option (def_inherited)`, described in "Checking Options" on page 198.

defer_codegen

Description Controls the inlining of functions that haven't been compiled yet.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma defer_codegen on | off | reset
```

Remarks This option allows inlining of inline and auto-inline functions that are called before their definition:

```
#pragma defer_codegen on
#pragma auto_inline on

extern void f();
extern void g();

main()
{
  f(); // will be inlined
  g(); // will be inlined
}

inline void f() {}
void g() {}
```

NOTE: The compiler will need more memory when this option is selected.

To check whether this option is on, use the `__option (defer_codegen)`, described in "Checking Options" on page 198.

define_section

Description Arranges object code into sections.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

`#pragma define_section` *sname istr* `[`*ustr*`]` `[`*addrmode*`]` `[`*accmode*`]`

Remarks This sophisticated and powerful pragma lets you arrange compiled object code into predefined sections and sections you define.

The parameters are:

- *sname*—identifier by which this user-defined section is referenced in the source.

  For example:

  `#pragma section` *sname* `begin`

  or

  `__declspec(`*sname*`)`

- *istr*—section name string for *initialized* data assigned to `sname`, such as `.data` (applies to uninitialized data if `ustr` is omitted)
- *ustr*—elf section name for *uninitialized* data assigned to `sname`
- *addrmode*—indicates how the section is addressed. It can be one of the following
  - `standard`—32-bit absolute address
  - `near_absolute`—16-bit absolute address
  - `far_absolute`—32-bit absolute address
  - `near_code`—16-bit offset from TP

  - `far_code`—32-bit offset from TP
  - `near_data`—16-bit offset from GP
  - `far_data`—32-bit offset from GP
- *accmode*—indicates the attributes of the section. It can be one of the following:
  - R—readable
  - RW—readable and writable
  - RX—readable and executable
  - RWX—readable, writable, and executable

The default value for `ustring` is the same as `istring`. The default value for `addrmode` is "`standard`". The default value for `accmode` is"`RWX`".

The compiler predefines the following common MIPS sections in absolute addressing mode (Table 6.1).

Table 6.1 MIPS predefined sections

```
#pragma define_section text ".text" far_absolute RX
#pragma define_section data ".data" ".bss" far_absolute RW
#pragma define_section sdata ".sdata" ".sbss" near_data RW
#pragma define_section const ".rodata" far_absolute R
```

In addition, the following are reserved for the C++ implementation (Table 6.2).

Table 6.2 MIPS predefined sections for C++

```
#pragma define_section exception ".exception" far_absolute R
#pragma define_section exceptlist ".exceptix" far_absolute R
#pragma define_section vtables ".vtables" far_absolute R
```

The compiler predefines the following common MIPS sections in PID addressing mode (Table 6.3).

Table 6.3 MIPS predefined sections in PID addressing mode

```
#pragma define_section text ".text" far_absolute RX
#pragma define_section data ".data" ".bss" far_data RW
#pragma define_section sdata ".sdata" ".sbss" near_data RW
#pragma define_section const ".rodata" far_absolute R
#pragma define_section exception ".exception" far_ data R
#pragma define_section exceptlist ".exceptix" far_ data R
#pragma define_section vtables ".vtables" far_ data R
```

The compiler predefines the following common MIPS sections in PIC addressing mode (Table 6.4).

Table 6.4 MIPS predefined sections in PIC addressing mode

```
#pragma define_section text ".text" far_code RX
#pragma define_section data ".data" ".bss" far_absolute RW
#pragma define_section sdata ".sdata" ".sbss" near_data RW
#pragma define_section const ".rodata" far_code R
#pragma define_section exception ".exception" far_absolute R
#pragma define_section exceptlist ".exceptix" far_absolute R
#pragma define_section vtables ".vtables" far_absolute R
```

The compiler predefines the following common V810/V830 sections (Table 6.5).

Table 6.5 Predefined sections for NEC V810 and V830 processors

```
#pragma define_section text ".text" far_code RX
#pragma define_section data ".data" ".bss" far_data RW
#pragma define_section sdata ".sdata" ".sbss" near_data RW
#pragma define_section itext ".itext" far_absolute RX
#pragma define_section const ".const" far_absolute R
#pragma define_section sconst ".sconst" near_absolute R
#pragma define_section sedata ".sedata" ".sebss" near_absolute RW
#pragma define_section sidata ".sidata" near_absolute RW
#pragma define_section cdata1 ".cdata1" far_absolute RW
#pragma define_section cdata2 ".cdata2" far_absolute RW
#pragma define_section cdata3 ".cdata3" far_absolute RW
#pragma define_section udata1 ".udata1" far_absolute RW
#pragma define_section udata2 ".udata2" far_absolute RW
#pragma define_section udata3 ".udata3" far_absolute RW
```

In addition, the following are reserved for the C++ implementation (Table 6.6):

Table 6.6 Predefined sections for NEC V810 and V830 processors (C++)

```
#pragma define_section exception ".exception" far_data R
#pragma define_section exceptlist ".exceptlist" far_data R
#pragma define_section vtables ".vtables" far_data R
#pragma define_section string ".string" far_data RW
#pragma define_section cstring ".cstring" far_data R
```

`#pragma define_section` can also be used to redefine the attributes of these existing sections:

1. You can force all data to be addressed using 16-bit absolute addresses using

```
#pragma define_section data ".data" near_absolute
```

2. You can force exception tables to be addressed using 32-bit TP-relative using

```
#pragma define_section exceptlist ".exceptlist" far_code
#pragma define_section exception ".exception" far_code
```

If you're going to do this, it is best to put these #pragmas in a prefix file or some other header that will be #included by all source files in your program.

NOTE: Any section that is user-defined in the compiler must be mapped to an appropriate segment in the ELF linker's **Section Mappings** settings panel:

NEV V800 Sections defined near_absolute must be assigned to a segment whose address is in the ranges `00000000:00007FFF` and `FFFF8000:FFFFFFFF`. Sections defined `near_code` or `far_code` must be assigned to the same segment as `.text`. Sections defined `near_data` or `far_data`'must be assigned to the same segment as `.data`

direct_destruction

Description Available for backwards-compatibility only.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma direct_destruction on | off | reset`

Remarks This option is available for backwards-compatibility only and is ignored. Use `#pragma exceptions` instead.

direct_to_som

Description Controls the generation of SOM object code.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma direct_to_som on | off | reset`

Remarks This pragma is available for Mac OS using C++ only.

This pragma lets you create SOM code directly in the CodeWarrior IDE. SOM is an integral part of OpenDoc. For more information, see *Targeting Mac OS*.

Note that when you turn on this pragma, Metrowerks C/C++ automatically turns on the **Enums Always Int** option in the C/C++ Language Panel, described in "Enumerated Types" on page 33.

This pragma corresponds to the **Direct to SOM** menu in the C/C++ Language Panel. Selecting **On** from that menu is like setting this pragma to on and setting the `SOMCheckEnviornment` pragma to `off`. Selecting **On with Environment Checks** from that menu is like setting both this pragma and `SOMCheckEnviornment` to on. Selecting **off** from that menu is like setting both this pragma and `SOMCheckEnviornment` to off. For more information on `SOMCheckEnviornment` see "SOMCheckEnvironment" on page 173.

To check whether this option is on, use the `__option (direct_to_SOM)`. See "Checking Options" on page 198. By default, this pragma is off.

disable_registers

Description Controls compatibility for the ANSI/ISO function `setjmp()`.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma disable_registers on | off | reset
```

Remarks If this option is on, the compiler turns off certain optimizations for any function that calls `setjmp()`. It disables global optimization and does not store local variables and arguments in registers. These changes ensure that all local variables will have up-to-date values.

NOTE: This option disables register optimizations in functions that use PowerPlant's `TRY` and `CATCH` macros but not in functions that use the ANSI-standard `try` and `catch` statements. The `TRY` and `CATCH` macros use `setjmp()`, but the `try` and `catch` statements are implemented at a lower level and do not use `setjmp()`.

For Mac OS, this pragma mimics a feature that's available in THINK C and Symantec C++. Use this pragma only if you're porting code that relies on this feature, since it drastically increases your code's size and decreases its speed. In new code, declare a variable to be `volatile` if you expect its value to persist across `setjmp()` calls.

This pragma does not correspond to any option in the PowerPC or NEC V800 settings panels. To check whether this option is on, use the `__option (disable_registers)`, described in "Checking Options" on page 198. By default, this option is off.

dollar_identifiers

Description Controls use of dollar signs ($) in identifiers.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma dollar_identifiers on | off | reset
```

Remarks

When this pragma is on, the compiler accepts dollar signs ($) in identifiers. When this option is off, the compiler issues an error if it encounters anything but underscores, alphabetic, and numeric characters in an identifier.

The default for this pragma is off.

This pragma does not correspond to any settings panel option. To check whether this option is on, use the `__option (dollar_identifiers)`, described in "Checking Options" on page 198.

dont_inline

Description

Controls the generation of inline functions.

Compatibility

This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma dont_inline on | off | reset
```

Remarks

If the pragma `dont_inline` is on, the compiler doesn't inline any function calls, even functions declared with the `inline` keyword or member functions defined within a class declaration. Also, it doesn't automatically inline functions, regardless of the setting of the `auto_inline` pragma, described in "auto_inline" on page 96. If this option is off, the compiler expands all inline function calls.

This pragma corresponds to the **Don't Inline** option of the **Inlining** menu the C/C++ Language Panel. To check whether this option is on, use `__option (dont_inline)`, described in "Checking Options" on page 198.

dont_reuse_strings

Description Specifies if each string literal should be stored separately in the string pool.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma dont_reuse_strings on | off | reset
```

Remarks If the pragma `dont_reuse_strings` is on, the compiler stores each string literal separately. If this pragma is off, the compiler stores only one copy of identical string literals. This pragma helps you save memory if your program contains lots of identical string literals which you do not modify.

For example, take this code segment:

```
char *str1="Hello";
char *str2="Hello"
*str2 = 'Y';
```

If this option is on, `str1` is `"Hello"` and `str2` is `"Yello"`. If this option is off, both `str1` and `str2` are `"Yello"`.

This pragma corresponds to the **Don't Reuse Strings** option in the C/C++ Language Panel. To check whether this option is on, use `__option (dont_reuse_strings)`, described in "Checking Options" on page 198.

ecplusplus

Description Controls the use of embedded C++ features.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma ecplusplus on | off | reset`

Remarks If this option is on the C++ compiler disables the non-EC++ features of ANSI C++ such as templates, multiple inheritance, and so on. See "C++ and Embedded Systems" on page 75 for more information on Embedded C++ support in CodeWarrrior C/C++.

This pragma doesn't correspond to any settings panel option. To check whether this option is on, use `__option (ecplusplus)`, described in "Checking Options" on page 198. By default this pragma is off.

EIPC_EIPSW

Description Controls the saving of processor information for interrupt functions.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma EIPC_EIPSW on|off|reset`

Remarks If this option is in effect when compiling an interrupt function, the compiler will also save or restore the `EIPC` and `EIPSW`. It is then safe to enable additional interrupts by calling `__EIEP()`.

This pragma doesn't correspond to any settings panel option. To check whether this option is on, use `__option (dont_reuse_strings)`, described in "Checking Options" on page 198.

enumsalwaysint

Description Specifies the size of enumerated types.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma enumsalwaysint on | off | reset`

Remarks When pragma `enumsalwaysint` is on, the C or C++ compiler makes an enumerated types the same size as an `int`. If an enumerated constant is larger than `int`, the compiler generates an error. When the pragma is off, the compiler makes an enumerated type the size of any integral type. It chooses the integral type with the size that most closely matches the size of the largest enumerated constant. The type could be as small as a `char` or as large as a `long int`.

For example:

```
enum SmallNumber { One = 1, Two = 2 };
  /* If enumsalwaysint is on, this type will
     be the same size as a char.
     If the pragma is off, this type will be
     the same size as an int. */

enum BigNumber
  { ThreeThousandMillion = 3000000000 };
  /* If enumsalwaysint is on, this type will
     be the same size as a long int.
     If this pragma is off, the compiler may
     generate an error. */
```

For more information on how the compiler handles enumerated types, see "Enumerated Types" on page 33.

This pragma corresponds to the **Enums Always Int** option in the C/C++ Language Panel. To check whether this option is on, use `__option (enumsalwaysint)`, described in "Checking Options" on page 198.

exceptions

Description Controls the availability of C++ exception handling.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma exceptions on | off | reset
```

Remarks

If you turn on this pragma, you can use the `try` and `catch` statements to perform exception handling. If your program doesn't use exception handling, turn this option off to make your program smaller.

You can throw exceptions across any code that's compiled by the CodeWarrior 8 (or later) Metrowerks C/C++ compiler with the **Enable C++ Exceptions** option turned on. You cannot throw exceptions across the following:

- Macintosh Toolbox function calls
- Libraries compiled with the **Enable C++ Exceptions** option turned off
- Libraries compiled with versions of the Metrowerks C/C++ compiler earlier than CodeWarrior 8
- Libraries compiled with Metrowerks Pascal or other compilers.

If you throw an exception across one of these, the code calls `terminate()` and exits.

This pragma corresponds to the **Enable C++ Exceptions** option in the C/C++ Language Panel. To check whether this option is on, use `__option (exceptions)`, described in "Checking Options" on page 198.

export

Description

Controls items to export from a module.

Compatibility

This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma export on | off | reset | list` *names*

Remarks This pragma applies to Mac OS programming only.

The pragma `export` gives you another way to export symbols besides using a `.exp` file. To export symbols with this pragma, choose **Use #pragma** from the **Export Symbols** menu in the PPC PEF or CFM68K settings panel. Then turn on this pragma to export variables and functions declared or defined in this file. If you choose any other option from the **Export Symbols** menu, the compiler ignores this pragma.

If you want to export all the functions and variables declared or defined within a certain range, use `#pragma export on` at the beginning of the range and use `#pragma export off` at the end of the range. If you want to export all the functions and variables in a list, use `#pragma export list`. If you want to export a single variable or function, use `__declspec(export)` at the beginning of the declaration

For example, this code fragment use `#pragma export on` and `off` to export the variable `w` and the functions `a1()` and `b1()`:

```
#pragma export on
int a1(int x, double y);
double b1(int z);
int w;
#pragma export off
```

This code fragment use `#pragma export list` to export the symbols:

```
int a1(int x, double y);
double b1(int z);
int w;
#pragma export list a1, b1, w
```

This code fragment uses `__declspec(internal)` to export the symbols:

```
__declspec(export) int a1(int x, double y);
__declspec(export) double b1(int z);
__declspec(export) int w;
```

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option(export)`, described in "Checking Options" on page 198.

extended_errorcheck

Description Controls the issuing of warnings for possible unintended logical errors.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma extended_errorcheck on | off | reset
```

Remarks If the pragma `extended_errorcheck` is on, the C compiler generates a warning (not an error) if it encounters one of the following:

- A non-`void` function that does not contain a `return` statement. For example, this would generate a warning:

```
main() /* assumed to return int */
{
  printf ("hello world\n");
}            /* WARNING: no return statement */
```

This would be OK:

```
void main()
{
  printf ("hello world\n");
}
```

- Assigning an integer or floating-point value to an `enum` type. For example:

```
enum Day { Sunday, Monday, Tuesday,
           Wednesday, Thursday,
           Friday, Saturday } d;

d = 5;          /* WARNING */
d = Monday;     /* OK */
d = (Day)3;     /* OK */
```

NOTE: Both of these are always errors in C++.

The C/C++ compiler generates a warning if it encounters this:

- An empty `return` statement (`return;`) in a function that is not declared `void`. For example, this code would generate a warning:

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err!=0) return;  // WARNING: Empty return statement

  // . . .
}
```

This would be OK:

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err!=0) return -1;  // OK

  // . . .
}
```

This pragma corresponds to the **Extended Error Checking** option in the C/C++ Warnings Panel. To check whether this option is on, use

`__option (extended_errorcheck)`, described in "Checking Options" on page 198.

far_code, near_code, smart_code

Description Specify the kind of addressing to use for executable code.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma far_code,
#pragma near_code,
#pragma smart_code
```

Remarks This pragma applies to Mac OS on 68K programming only.

These pragmas determine what kind of addressing the compiler uses to refer to functions:

- `#pragma far_code` always generates 32-bit addressing, even if 16-bit addressing can be used
- `#pragma near_code` always generates 16-bit addressing, even if data or instructions are out of range.
- `#pragma smart_code` generates 16-bit addressing whenever possible and uses 32-bit addressing only when necessary.

For more information on these code models, see the *CodeWarrior User's Guide*.

These pragmas correspond to the **Code Model** option in the 68K Processor settings panel. The default is `#pragma smart_code`.

far_data

Description Controls the use of 32-bit addressing to refer to global data.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma far_data on | off | reset
```

Remarks

If this pragma is on, you can have any amount of global data since the compiler uses 32-bit addressing to refer to globals instead of 16-bit addressing. Your program will also be slightly bigger and slower. this pragma is off, your global data is stored as near data and add to the 64K limit on near data.

This pragma corresponds to the **Far Data** option in the 68K Processor settings panel. To check whether this option is on, use `__option (far_data)`, described in "Checking Options" on page 198.

far_strings

Description

Controls the use of 32-bit addressing to refer to string literals.

Compatibility

This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma far_strings on | off | reset
```

Remarks

If this pragma is on, you can have any number of string literals since the compiler uses 32-bit addressing to refer to string literals, instead of 16-bit addressing. Your program will also be slightly bigger and slower. If this pragma is off, your string literals are stored as near data and add to the 64K limit on near data.

This pragma corresponds to the **Far String Constants** option in the 68K Processor settings panel. To check whether this option is on, use `__option (far_strings)`, described in "Checking Options" on page 198.

far_vtables

Description Controls the use of 32-bit addressing for C++ virtual function tables.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma far_vtables on | off | reset
```

Remarks This pragma applies to Mac OS on 68K programming only.

A class with virtual function members has to create a virtual function dispatch table in a data segment. If this pragma is on, that table can be any size since a the compiler uses 32-bit addressing to refer to the table, instead of 16-bit addressing. Your program will also be slightly bigger and slower. If this pragma is off, the table is stored as near data and adds to the 64K limit on near data.

This pragma corresponds to the **Far Method Tables** option in the 68K Processor settings panel. To check whether this option is on, use `__option (far_vtables)`, described in "Checking Options" on page 198.

faster_pch_gen

Description Controls the performance of precompiled header generation.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma faster_pch_gen on | off | reset
```

Remarks Turning this pragma on can make writing a precompiled header much faster (depending on the header structure). Precompiled header files that are generated with this option are slightly bigger. The default is off.

This pragma does not correspond to any option in any settings panel. To check whether this option is on, use the `__option (faster_pch_gen)`, described in "Checking Options" on page 198.

force_active

Description Controls how "dead" functions are linked.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma force_active on | off | reset
```

Remarks This pragma applies to Mac OS on 68K programming only.

If this option is on, the linker will not strip the following functions out of the finished application, even if the functions are never called in the program.

In Macintosh code, this option is off by default.

This pragma does not correspond to any option in any settings panel. To check whether this option is on, use the `__option (force_active)`, described in "Checking Options" on page 198.

fourbyteints

Description Controls the size of the `int` data type.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma fourbyteints on | off | reset
```

Remarks When this option is on, the size of an `int` is 4 bytes. When this option is off, the size of an `int` is 2 byes.

This pragma corresponds to the **4-Byte Ints** option in the **68K Processor** settings panel. To check whether this option is on, use `__option (fourbyteints)`, described in "Checking Options" on page 198.

NOTE: Whenever possible, set this option from the settings panel and not from a pragma. If you must set it from a pragma, place the pragma at the beginning of your program, before you include any files or declare any functions or variables.

fp_contract

Description Controls the use of special floating point instructions to improve performance.

Compatibility This pragma is compatible with the following platform targets:

68K	**PowerPC**	NEC V800	Intel x86	MIPS

Prototype

```
#pragma fp_contract on | off | reset
```

Remarks This pragma applies to PowerPC programming only.

If this pragma is on, the compiler uses such PowerPC instructions as `FMADD`, `FMSUB`, and `FNMAD` to speed up floating-point computations. However, certain computations give unexpected results when this pragma is on. For example:

```
register double A, B, C, D, Y, Z;
register double T1, T2;

A = C = 2.0e23;
B = D = 3.0e23;

Y = (A * B) - (C * D);
printf("Y = %f\n", Y);
/* prints 21267700587560961875633692999968.000000 */
```

```
T1 = (A * B);
T2 = (C * D);
Z = T1 - T2;
printf("Z = %f\n", Z);  /* prints 0.000000 */
```

When this option is off, `Y` and `Z` have the same value.

This pragma corresponds to the **Use FMADD & FMSUB** option in the PPC Processor settings panel. To check whether this option is on, use `__option (fp_contract)`, described in "Checking Options" on page 198.

fp_pilot_traps

Description Controls floating point code generation for Palm OS.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma fp_pilot_traps on | off | reset
```

Remarks This pragma controls floating point code generation. If on, the compiler makes references to Palm OS library routines to perform floating point operations.

function

Description Ignored but included for compatibility with Microsoft compilers.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	**Intel x86**	MIPS

Prototype

```
#pragma function( funcname1, funcname2, ... )
```

Remarks Ignored. Included for compatibility with Microsoft.

global_optimizer, optimization_level

Description Controls optimization.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma global_optimizer on | off | reset
#pragma optimization_level 1 | 2 | 3 | 4 | 5
```

Remarks These pragmas control the global optimizer performs. To turn the global optimizer on and off, use the pragma `global_optimizer`. To choose which optimizations the global optimizer performs, use the pragma `optimization_level` with an argument from `1` to `5`. The higher the argument, the more optimizations that the global optimizer performs. If the global optimizer is turned off, the compiler ignores the pragma `optimization_level`.

For more information on the optimization the compiler performs for each optimization level, refer to the *Targeting* manual for the platform you're developing for.

These pragmas correspond to the options in the **Global Optimizations** settings panel. To check whether the global optimizer is on, use `__option (global_optimizer)`, described in "Checking Options" on page 198.

IEEEdoubles

Description Specifies the size of the `double` type.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma IEEEdoubles on | off | reset`

Remarks This option, along with the **68881 Codegen** option, specifies the length of a `double`. The table below shows how these options work:

If `IEEEDoubles` **is…**	**and** `code68881` **is…**	**Then a** `double` **is this size…**
on	on or off	64 bits
off	off	80 bits
off	on	96 bits

This pragma corresponds to the **8-Byte Doubles** option in the 68K Processor settings panel. To check whether this option is on, use `__option (IEEEdoubles)`, described in "Checking Options" on page 198.

NOTE: Whenever possible, set this option from the settings panel and not from a pragma. If you must set it from a pragma, place the pragma at the beginning of your program, before you include any files or declare any functions or variables.

ignore_oldstyle

Description Controls the recognition of function declaration that follow the convention before ANS/ISO C.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma ignore_oldstyle on | off | reset`

Remarks If pragma `ignore_oldstyle` is on, the compiler ignores old-style function declarations and lets you prototype a function any way you want. In old-style declarations, you don't specify the types of

the arguments in the argument list but on separate lines. It's the style of declaration used in the first edition of *The C Programming Language* (Prentice Hall) by Kernighan and Ritchie.

For example, this code defines a prototype for a function with an old-style declaration:

```
int f(char x, short y, float z);

#pragma ignore_oldstyle on

f(x, y, z)
char x;
short y;
float z;
{
  return (int)x+y+z;
}

#pragma ignore_oldstyle reset
```

This pragma does not correspond to an option in any settings panel. By default this option is off. To check whether this option is on, use `__option (ignore_oldstyle)`, described in "Checking Options" on page 198.

import

Description Controls or specifies the availability of imported symbols.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma import on | off | reset | list` *names*

Remarks This pragma applies to Mac OS CFM programming only.

This pragma lets you import variables and functions that are in other fragments. Use this to import symbols that have been exported with the `export` pragma, an `.exp` file, or the **Export Symbols** menu in the CFM68K and PPC PEF settings panel.

If you want to import all the functions and variables declared or define within a certain range, use `#pragma import on` at the beginning of the range and use `#pragma import off` at the end of the range. If you want to import all the functions and variables in a list, use `#pragma import list`. If you want to import a single variable or function, use `__declspec(external)` at the beginning of the declaration

For example, this code fragment use `#pragma import on` and `off` to import the variable `w` and the functions `a1()` and `b1()`:

```
#pragma import on
int a1(int x, double y);
double b1(int z);
int w;
#pragma import off
```

This code fragment use `#pragma import list` to import the symbols:

```
int a1(int x, double y);
double b1(int z);
int w;
#pragma import list a1, b1, w
```

And this code fragment uses `__declspec(import)` to import the symbols:

```
__declspec(import) int a1(int x, double y);
__declspec(import) double b1(int z);
__declspec(import) int w;
```

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option (import)`, described in "Checking Options" on page 198.

init_seg

Description Controls the order in which initialization code is executed.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
pragma init_seg( compiler | lib | user | "name ")
```

Remarks This pragma controls the order in which initialization code is executed.The initialization code for a C++ compiled module calls constructors for any statically declared objects. For C, no initialization code is generated.

The order of initialization is

1.compiler

2.lib

3.user

If you specify the name of a segment, a pointer to the initialization code is placed in the designated segment. In this case, the initialization code is not called automatically: it's up to you to call it explicitly.

inline_depth

Description Controls how deeply inline function calls are expanded.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma inline_depth(n)
#pragma inline_depth(smart)
```

Remarks Sets the number of passes used to expand inline function calls. The number *n* is an integer from `0` to `1024` or the `smart` specifier.

The `smart` specifier is the default mode, with 4 passes where the passes 2-4 are limited to small inline functions. All inlineable functions are expanded if `inline_depth` is set to 1-1024.

The pragmas `dont_inline` and `always_inline` override this pragma.

inline_intrinsics

Description Controls the inlining of intrinsic functions.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	**Intel x86**	MIPS

Prototype

```
#pragma inline_instrinsics on | off | reset
```

Remarks When this option is on, the compiler directly generates intrinsic functions without generating a function call.

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option(inline_intrinsics)`, described in "Checking Options" on page 198.

internal

Description Controls the availability of symbols outside a module.

Compatibility This pragma is compatible with the following platform targets:

68K	**PowerPC**	NEC V800	Intel x86	MIPS

Prototype

```
#pragma internal on | off | reset | list names
```

Remarks This pragma applies to Mac OS CFM programming only.

This pragma lets you specify that certain variables and functions are internal and not imported. The compiler generates smaller and

faster code when it calls an internal function, even if you declared it as extern.

If you want to declare all the functions and variables declared or define within a certain range as internal, use `#pragma internal on` at the beginning of the range and use `#pragma internal off` at the end of the range. If you want to declare all the functions and variables in a list as internal, use `#pragma internal list`. If you want to declare a single variable or function as internal, use `__declspec(internal)` at the beginning of the declaration.

For example, this code fragment use `#pragma internal on` and `off` to declare the variable `w` and the functions `a1()` and `b1()` as internal:

```
#pragma internal on
int a1(int x, double y);
double b1(int z);
int w;
#pragma internal off
```

This code fragment uses `#pragma internal list` to declare the symbols as internal:

```
int a1(int x, double y);
double b1(int z);
int w;
#pragma internal list a1, b1, w
```

And this code fragment uses `__declspec(internal)` to declare the symbols as internal:

```
__declspec(internal) int a1(int x, double y);
__declspec(internal) double b1(int z);
__declspec(internal) int w;
```

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option (internal)`, described in "Checking Options" on page 198.

interrupt

Description Controls the compilation of object code for interrupt routines.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	**NEC V800**	Intel x86	MIPS

Prototype

```
#pragma interrupt on|off|reset
```

Remarks When this option is on, the compiler generates a special prologue and epilogue for these functions: all modified registers (both non-volatile and scratch registers) are saved or restored, and the function returns via `RETI` instead of `JMP [LP]`.

For convenience, the compiler will also mark any interrupt function so that the linker does not dead-strip it.

k63d

Description Controls special code generation for AMD K6 3D extensions.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	**Intel x86**	MIPS

Prototype

```
#pragma k63d on | off | reset
```

Remarks This pragma tells the x86 compiler to generate code for AMD K6 3D extensions. This option causes the compiler to generate code that will only run on processors that are equipped with the circuitry needed to execute the specialized 3D instructions.

This pragma corresponds to the **K6 3D Favored** option in the **Extended Instruction Set** menu of the **x86 CodeGen** settings panel.

NOTE: This `#pragma` generates code that is not compatible with the Intel Pentium class of microprocessors.

To learn more about this pragma, read the *Targeting Win32* manual.

k63d_calls

Description Controls use of AMD K6 3D calling conventions.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	**Intel x86**	MIPS

Prototype

```
#pragma k63d_calls on | off | reset
```

Remarks This pragma tells the x86 compiler to generate code for AMD K6 3D and Intel MMX extensions. This option causes the compiler to generate code that will only run on processors that are equipped with the circuitry needed to execute these specialized instruction sets. This pragma generates code that requires fewer register operations at mode switching time.

This pragma corresponds to the **MMX + K6 3D** option in the **Extended Instruction Set** menu of the **x86 CodeGen** settings panel.

To learn more about this pragma, read the *Targeting Win32* manual.

lib_export

Description Controls the recognition of the `export`, `import`, and `internal` pragmas.

Compatibility This pragma is compatible with the following platform targets:

68K	**PowerPC**	NEC V800	Intel x86	MIPS

Prototype `#pragma lib_export on | off | reset`

Remarks This pragma applies to Mac OS CFM programming only.

If this pragma is off, the compiler ignores the pragmas export, import, and internal. It is available for compatibility with previous versions of the compiler. It corresponds to the `__declspec(lib_export)` type qualifier, described in "ANSI Keywords Only" on page 41. To check whether this option is on, use `__option (lib_export)`, described in "Checking Options" on page 198.

This pragma does not correspond to an option in any settings panel.

longlong

Description Controls the availability of the `long long` type.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma longlong on | off | reset`

Remarks When the `longlong` pragma is on, the C or C++ compiler lets you define a 64-bit integer with the type specifier `long long`. This is twice as large as a long int, which is a 32-bit integer. A `long long` can hold values from `-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807`. An `unsigned long long` can hold values from `0` to `18,446,744,073,709,551,615`.

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option (longlong)`, described in "Checking Options" on page 198.

longlong_enums

Description Controls whether or not enumerated types are the size of the `long long` type.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma longlong_enums on | off | reset
```

Remarks This pragma lets you use enumerators that large enough to be `long long` integers. It's ignored if the `enumsalwaysint` pragma is on (described in "enumsalwaysint" on page 115).

For more information on how the compiler handles enumerated types, see "Enumerated Types" on page 33.

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option (longlong_enums)`, described in "Checking Options" on page 198. By default, this option is on.

macsbug, oldstyle_symbols

Description Control the generation of debugger data for MacsBug.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma macsbug on | off | reset
#pragma oldstyle_symbols on | off | reset
```

Remarks This pragma applies to Mac OS on 68K programming only.

These pragmas let you choose how the compiler generates Macsbug symbols. Many debuggers, including Metrowerks debugger, use Macsbug symbols to display the names of functions and variables. The pragma `macsbug` lets you turn on and off Macsbug generation. The pragma `oldstyle_symbols` lets you choose which type of

symbols to generate. The table below shows how these pragmas work:

To do this…	Use these pragmas…
Do not generate Macsbug symbols	`#pragma macsbug on`
Generate old style Macsbug symbols	`#pragma macsbug on` `#pragma oldstyle_symbols on`
Generate new style Macsbug symbols	`#pragma macsbug on` `#pragma oldstyle_symbols off`

These pragmas corresponds to **MacsBug Symbols** option in the 68K Linker settings panel. To check whether the `macsbug` pragma option is on, use `__option (macsbug)`, described in "Checking Options" on page 198. To check whether the old style pragma is on, use `__option (oldstyle_symbols)` described in "Checking Options" on page 198.

mark

Description Adds an item to the Function pop-up menu in the IDE's editor window.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma mark itemName
```

Remarks This pragma adds *itemName* to the source file's Function pop-up menu. If you open the file in the CodeWarrior Editor and select the item from the Function pop-up menu, the editor brings you to the pragma. Note that if the pragma is inside a function definition, the item will not appear in the Function pop-up menu.

If itemName begins with "--" a menu separator appears in the IDE's function pop-up menu:

```
#pragma mark --
```

This pragma does not correspond to an option in any settings panel.

microsoft_exceptions

Description Controls the use of Microsoft C++ exception handling.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	**Intel x86**	MIPS

Prototype

```
#pragma microsoft_exceptions on | off | reset
```

Remarks This pragma tells the x86 compiler to generate exception handling code that is compatible with Microsoft C++ exception handling code.

To check whether this option is on, use `__option(microsoft_exceptions)`, described in "Checking Options" on page 198.

microsoft_RTTI

Description Controls the use of Microsoft C++ runtime type information.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	**Intel x86**	MIPS

Prototype

```
#pragma microsoft_RTTI on | off | reset
```

Remarks This pragma tells the x86 compiler to generate runtime type information that is compatible with Microsoft C++.

To check whether this option is on, use `__option(microsoft_RTTI)`, described in "Checking Options" on page 198.

mmx

Description Controls special code generation Intel MMX extensions.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	**Intel x86**	MIPS

Prototype

```
#pragma mmx on | off | reset
```

Remarks This pragma tells the x86 compiler to generate code for Intel MMX extensions. This option causes the compiler to generate code that will only run on processors that are equipped with the circuitry needed to execute the more than 50 specialized MMX instructions.

This pragma corresponds to the **MMX** option in the **Extended Instruction Set** menu of the **x86 CodeGen** settings panel. To check whether this option is on, use `__option (mmx)`, described in "Checking Options" on page 198.

To learn more about this pragma, read the *Targeting Win32* manual.

mmx_call

Description Controls the use of MMX calling conventions.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	**Intel x86**	MIPS

Prototype

```
#pragma mmx_call on | off | reset
```

Remarks When this pragma is on, the compiler favors the use of MMX calling conventions.

To check whether this option is on, use `__option (mmx_call)`, described in "Checking Options" on page 198.

To learn more about this pragma, read the *Targeting Win32* manual.

mpwc

Description Controls the use Apple's MPW C calling conventions.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma mpwc on | off | reset
```

Remarks This pragma applies to Mac OS on 68K processors only.

When the pragma `mpwc` is on, the compiler does the following to be compatible with MPW C's calling conventions:

- Passes any integral argument that is smaller than 2 bytes as a sign-extended `long integer`. For example, the compiler converts this declaration:

```
int MPWfunc ( char a, short b, int c, long d, char *e );
```

- To this:

```
long MPWfunc( long a, long b, long c, long d, char *e );
```

- Passes any floating-point arguments as a `long double`. For example, the compiler converts this declaration:

```
void MPWfunc( float a, double b, long double c );
```

- To this:

```
void MPWfunc( long double a, long double b, long double c );
```

- Returns any pointer value in `D0` (even if the pragma `pointers_in_D0` is off).
- Returns any 1-byte, 2-byte, or 4-byte structure in `D0`.

- If the **68881 Codegen** option is on, returns any floating-point value in `FP0`.

This pragma corresponds to the **MPW C Calling Convention** option in the 68K Processor settings panel. To check whether this option is on, use `__option (mpwc)`, described in "Checking Options" on page 198.

mpwc_newline

Description Controls the use new line character convention used by Apple's MPW C.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma mpwc_newline on | off | reset
```

Remarks If you turn on the pragma `mpwc_newline`, the compiler uses the MPW conventions for the `'\n'` and `'\r'` characters. If this pragma is off, the compiler uses the Metrowerks C and C++ conventions for these characters.

In MPW, `'\n'` is a Carriage Return (0x0D) and `'\r'` is a Line Feed (0x0A). In Metrowerks C and C++, they're reversed: `'\n'` is a Line Feed and `'\r'` is a Carriage Return.

If you want to turn this pragma on, be sure you use the ANSI C and C++ libraries that were compiled with this option on. The 68K versions of these libraries are marked with an NL; for example, `MSL C.68K (NL_2i).Lib`. The PowerPC versions of these libraries are marked with NL; for example, `MSL C.PPC (NL).Lib`.

If you turn this pragma on and use the standard ANSI C and C++ libraries, you won't be able to read and write `'\n'` and `'\r'` properly. For example, printing `'\n'` brings you to the beginning of the current line instead of inserting a new line.

This pragma corresponds to the **Map Newlines to CR** option in the C/C++ Language Panel. To check whether this option is on, use `__option (mpwc_newline)`, described in "Checking Options" on page 198.

mpwc_relax

Description Controls the compatibility of the `char*` and `unsigned char*` types.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma mpwc_relax on | off | reset
```

Remarks When you turn on this pragma, the compiler treats `char*` and `unsigned char*` as the same type. This option is especially useful if you're using code written before the ANSI C standard. This old source code frequently used these types interchangeably.

This option has no effect on C++ source code.

This pragma may be used to relax function pointer checking:

```
#pragma mpwc_relax on
extern void f(char *);
extern void(*fp1)(void *) = &f;          // error but allowed
extern void(*fp2)(unsigned char *) = &f; // error but allowed
```

This pragma corresponds to the **Relaxed Pointer Type Rules** option in the C/C++ Language Panel. To check whether this option is on, `__option (mpwc_relax)`, described in "Checking Options" on page 198.

no_register_coloring

Description Controls the use of a register to hold the values of more than one variable.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma no_register_coloring on | off | reset
```

Remarks When the `no_register_coloring` pragma is off, the compiler performs register coloring. In this optimization, the compiler lets two or more variables share a register: it assigns different variables or parameters to the same register if you do not use the variables at the same time. In this example, the compilers could place `i` and `j` in the same register:

```
short i;
int j;

for (i=0; i<100;  i++) {   MyFunc(i);  }
for (j=0; j<1000; j++) {   OurFunc(j); }
```

However, if a line like the one below appears anywhere in the function, the compiler would realize that you're using `i` and `j` at the same time and place them in different registers:

```
int k = i + j;
```

If register coloring is on while you debug your project, it may appear as though there's something wrong with the variables sharing a register. In the example above, `i` and `j` would always have the same value. When `i` changes, `j` changes in the same way. When `j` changes, `i` changes in the same way. To avoid this confusion while debugging, turn off register coloring or declare the variables you want to watch as volatile.

The pragma corresponds to the **Global Register Allocation** option in the 68K Processor settings panel. To check whether this option is on, use `__option (no_register_coloring)`, described in "Checking Options" on page 198. By default, this option is off.

NOTE: To turn off register coloring in code for a PowerPC Macintosh, use the statement `#pragma global_optimizer off`. For more information, see "global_optimizer, optimization_level" on page 127.

See also "register_coloring" on page 161.

once

Description Specifies if a header file may be included more than once in the same source file.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma once [ on | off ]
```

Remarks Use this pragma to ensure that the compiler includes header files only once in a source file. This pragma is especially useful in precompiled header files.

There are two versions of this pragma: `#pragma once` and `#pragma once on`. Use `#pragma once` in a header file to ensure that the header file is included only once in a source file. Use `#pragma once on` in a header file or source file to insure that *any* file is included only once in a source file.

This pragma does not correspond to an option in any settings panel. By default this option is off.

only_std_keywords

Description Controls the use of ANSI/ISO keywords.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma only_std_keywords on | off | reset`

Remarks The C/C++ compiler recognizes additional reserved keywords. If you're writing code that must follow the ANSI standard strictly, turn on the pragma `only_std_keywords`. For more information, see "ANSI Keywords Only" on page 41.

This pragma corresponds to the **ANSI Keywords Only** option in the C/C++ Language Panel. To check whether this option is on, use `__option (only_std_keywords)`, described in "Checking Options" on page 198.

opt_common_subs

Description Controls the use of common subexpression optimization.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma opt_common_subs on | off | reset`

Remarks When this option is on, the compiler replaces similar redundant expressions with a single expression. For example, if two statements in a function both use the expression

```
a * b * c + 10
```

the compiler generates object code that computes the expression only once and applies the resulting value to both statements.

The pragma doesn't correspong to an option in any settings panel. To check whether this option is on, use `__option (opt_common_subs)`, described in "Checking Options" on page 198.

opt_dead_assignments

Description Controls the use of dead store optimization.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma opt_dead_assignments on | off | reset
```

Remarks When this option is on, the compiler removes assignments to variables if the variables are not used before being reassigned.

The pragma doesn't correspong to an option in any settings panel. To check whether this option is on, use `__option(opt_dead_assignments)`, described in "Checking Options" on page 198.

opt_dead_code

Description Controls the use of dead code optimization.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma opt_dead_code on | off | reset
```

Remarks When this option is on, the compiler removes statements that, logically, will never be executed or is never referred to by other statements.

The pragma doesn't correspong to an option in any settings panel. To check whether this option is on, use `__option(opt_dead_code)`, described in "Checking Options" on page 198.

opt_lifetimes

Description Controls the use of lifetime analysis optimization.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma opt_lifetimes on | off | reset`

Remarks When this option is on, the compiler uses the same processor register for different variables in the same routine if the variables aren't used in the same statement.

The pragma doesn't correspong to an option in any settings panel. To check whether this option is on, use `__option(opt_lifetimes)`, described in "Checking Options" on page 198.

opt_loop_invariants

Description Controls the use of loop invariant optimization.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma opt_loop_invariants on | off | reset`

Remarks When this option is on, the compiler moves computations that don't change on the inside of a loop to the outside of a loop to improve the loop's speed.

The pragma doesn't correspong to an option in any settings panel. To check whether this option is on, use `__option(opt_loop_invariants)`, described in "Checking Options" on page 198.

opt_propagation

Description Controls the use of copy and constant propagation optimization.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma opt_propagation on | off | reset
```

Remarks When this option is on, the compiler replaces multiple occurrences of one variable with a single occurrence.

The pragma doesn't correspong to an option in any settings panel. To check whether this option is on, use `__option(opt_proagation)`, described in "Checking Options" on page 198.

opt_strength_reduction

Description Controls the use of strength reduction optimization.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma opt_strength_reduction on | off | reset
```

Remarks When this option is on the compiler replaces multiplication instructions that are inside loops with addition instructions to speed up the loops.

The pragma doesn't correspong to an option in any settings panel. To check whether this option is on, use `__option(opt_strength_reduction)`, described in "Checking Options" on page 198.

opt_unroll_loops

Description Controls the use of loop unrolling optimization.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma opt_unroll_loops on | off | reset
```

Remarks When this option is on the compiler places multiple copies of a loop's statements inside a loop to improve its speed.

The pragma doesn't correspong to an option in any settings panel. To check whether this option is on, use `__option (opt_unroll_loops)`, described in "Checking Options" on page 198.

opt_vectorize_loops

Description Controls the use of loop vectorizing optimization.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma opt_vectorize_loops on | off | reset
```

Remarks When this option is on the compiler improves loop performance.

The pragma doesn't correspong to an option in any settings panel. To check whether this option is on, use `__option (opt_vectorize_loops)`, described in "Checking Options" on page 198.

optimization_level

See the pragma `global_optimizer`, described in "global_optimizer, optimization_level" on page 127.

optimize_for_size

Description Controls optimization to reduce the size of object code.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma optimize_for_size on | off | reset
```

Remarks This option lets you choose what the compiler does when it must decide between creating small code or fast code. If this option is on, the compiler creates smaller object code at the expense of speed. If this option is off, the compiler creates faster object code at the expense of size.

Most significantly if this option is on, the compiler ignores the `inline` directive, and generates function calls to call any function declared `inline`.

The pragma corresponds to the **Optimize for Size** option **Global Optimizations** settings panel. To check whether this option is on, use `__option (optimize_for_size)`, described in "Checking Options" on page 198.

oldstyle_symbols

See "macsbug, oldstyle_symbols" on page 137 for information about this pragma.

pack

Description Controls the alignment of data structures.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma pack( [`*n* `|` `push,` *n* `|` `pop] )`

Remarks Sets the packing alignment for data structures. It affects all data structures declared after this pragma until you change it again with another `pack` pragma.

This pragma…	Does this…
`#pragma pack(n)`	Sets the alignment modulus to *n*, where *n* may be `1`, `2`, `4`, `8` or `16`. For MIPS compilers, if n is 0, structure alignment is reset to the default setting.
`#pragma pack(push, n)`	Pushes the current alignment modulus on a stack, then sets it to *n*, where *n* may be `1`, `2`, `4`, `8` or `16`. Use `push` and `pop` when you need a specific modulus for some declaration or set of declarations, but do not want to disturb the default setting. This form is not supported by the MIPS compilers.
`#pragma pack(pop)`	Pops a previously pushed alignment modulus from the stack. This form is not supported by the MIPS compilers.
`#pragma pack()`	For x86 compilers, resets alignment modulus to the value specified in the x86 CodeGen settings panel. For MIPS compilers, resets structure alignment to the default setting.

This pragma corresponds to the **Byte Alignment** option in the **x86 CodeGen** settings panel.

parameter

Description Specifies the use of registers to pass parameters.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma parameter` *return-reg func-name*`(`*param-regs*`)`

Remarks This pragma applies to 68K programming only.

The compiler passes the parameters for the function *func-name* in the registers specified in *param-regs* instead of the stack, and returns any return value in the register *return-reg*. Both *return-reg* and *param-regs* are optional.

Here are some samples:

```
#pragma parameter __D0 Gestalt(__D0, __A1)
#pragma parameter __A0 GetZone
#pragma parameter HLock(__A0)
```

When you define the function, you need to specify the registers right in the parameter list, as described in the appropriate Targeting manual.

This pragma does not correspond to an option in any settings panel.

pcrelstrings

Description Controls the storage and reference of string literals from the program counter.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma pcrelstrings on | off | reset`

Remarks If this option is on, the compiler stores the string constants used in a local scope in the code segment and addresses these strings with PC-relative instructions. If this option is off, the compiler stores all

string constants in the global data segment. Regardless of how this option is set, the compiler stores string constants used in the global scope in the global data segment. For example:

```
#pragma pcrelstrings on
int foo(char *);

int x = f("Hello");  // "Hello" is allocated in
                     // the global data segment
int bar()
{
  return f("World"); //"World" is allocated in the code segment
}                    // (pc-relative)
```

Strings in C++ initialization code are always allocated in the global data segment.

NOTE: If you turn the `pool_strings` pragma on, the compiler ignores the setting of the `pcrelstrings` pragma.

This pragma corresponds to the **PC-Relative Strings** option in the 68K Processor settings panel. To check whether this option is on, use `__option (pcrelstrings)`, described in "Checking Options" on page 198. By default, this option is off.

peephole

Description Controls the use peephole optimization.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma peephole on | off | reset
```

Remarks If this pragma is on, the compiler performs peephole optimizations, which are small local optimizations that eliminate some compare instructions and improve branch sequences.

This pragma corresponds to the **Peephole Optimizer** option in the PPC Processor settings panel. To check whether this option is on, use `__option (peephole)`, described in "Checking Options" on page 198.

pointers_in_A0, pointers_in_D0

Description Controls which calling convention to use.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma pointers_in_A0
#pragma pointers_in_D0
```

Remarks These pragmas are available for Mac OS on 68K processors only.

These pragmas let you choose between two calling conventions: the convention for MPW and Macintosh Toolbox routines and the convention for Metrowerks C and C++ routines. In the MPW and Macintosh Toolbox calling convention, functions return pointers in the register `D0`. In the Metrowerks C and C++ convention, functions return pointers in the register `A0`.

When you declare functions from the Macintosh Toolbox or a library compiled with MPW, use the pragma `pointers_in_D0`. After you declare those functions, use the pragma `pointers_in_A0` to start declaring or defining Metrowerks C and C++ functions.

In Listing 6.3, the Toolbox functions in `Sound.h` return pointers in `D0` and the user-defined functions in `Myheader.h` use `A0`.

Listing 6.3 Using #pragma pointers_in_A0 and #pragma pointers_in_D0

```
#pragma pointers_in_D0 // set for Toolbox calls
#include <Sound.h>
```

```
#pragma pointers_in_A0 // set for my own routines
#include "Myheader.h"
```

The pragmas `pointers_in_A0` and `pointers_in_D0` have much the same meaning as `d0_pointers` and are available for backwards compatibility. The pragma `pointers_in_A0` corresponds to `#pragma d0_pointers off` and the pragma `pointers_in_D0` corresponds to `#pragma d0_pointers on`. The pragma `d0_pointers` is recommended for new code since it supports the `reset` argument. For more information, see "d0_pointers" on page 102.

This pragma does not correspond to any option in the settings panel. To check whether this option is on, use the `__option (d0_pointers)`, described in "Checking Options" on page 198.

pool_data

Description Controls how data is stored.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma pool_data on | off | reset
```

Remarks This pragma is available for embedded PowerPC programming only.

If this pragma is on, the compiler optimizes pooled data. The pragma must be used before the function you want it apply to.

This pragma corresponds to the **Pool Data** option in the PPC Processor settings panel. To check whether this option is on, use `__option (pool_data)`, described in "Checking Options" on page 198.

pool_strings

Description Controls how string literals are stored.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma pool_strings on | off | reset
```

Remarks If the pragma `pool_strings` in the C/C++ Language Panel is on, the compiler collects all string constants into a single data object so your program needs one TOC entry for all of them. If this pragma is off, the compiler creates a unique data object and TOC entry for each string constant. Turning this pragma on decreases the number of TOC entries in your program but increases your program's size, since it uses a less efficient method to store the string's address.

This pragma is especially useful if your program is large and has many string constants or uses the Metrowerks Profiler.

NOTE: If you turn the `pool_strings` pragma on, the compiler ignores the setting of the `pcrelstrings` pragma.

This pragma corresponds to the **Pool Strings** option in the C/C++ Language Panel. To check whether this option is on, use `__option(pool_strings)`, described in "Checking Options" on page 198.

pop, push

Description Save and restore pragma settings.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma push
#pragma pop
```

Remarks The pragma `push` saves all the current pragma settings. The pragma `pop` restores all the pragma settings to what they were at the last `push` pragma. For example, see Listing 6.4.

Listing 6.4 push and pop example

```
#pragma far_data on
#pragma pointers_in_A0
#pragma push   // push all compiler options
#pragma far_data off
#pragma pointers_in_D0
   // pop restores "far_data" and "pointers_in_A0"
#pragma pop
```

These pragmas are available so you can use MacApp with Metrowerks C and C++. If you're writing new code and need to set a pragma option to its original value, use the `reset` argument, described in "Pragma Syntax" on page 88.

precompile_target

Description Specifies the file name for a precompiled header file.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma precompile_target filename
```

Remarks

NOTE: This pragma is not supported on Be OS.

This pragma specifies the filename for a precompiled header file. If you don't specify the filename, the compiler gives the precompiled header file the same name as its source file.

Filename can be a simple filename or an absolute pathname. If *filename* is a simple filename, the compiler saves the file in the same folder as the source file. If *filename* is a path name, the compiler saves the file in the specified folder.

Listing 6.5 shows sample source code from the MacHeaders precompiled header source file. By using the predefined symbols `__cplusplus` and `powerc` and the pragma `precompile_target`, the compiler can use the same source code to create different precompiled header files for C and C++, 680x0 and PowerPC.

Listing 6.5 Using #pragma precompile_target filename

```
#ifdef __cplusplus
 #ifdef powerc
  #pragma precompile_target "MacHeadersPPC++"
 #else
  #pragma precompile_target "MacHeaders68K++"
 #endif
#else
 #ifdef powerc
  #pragma precompile_target "MacHeadersPPC"
 #else
  #pragma precompile_target "MacHeaders68K"
 #endif
#endif
```

profile

Description Controls the generation of extra object code for use with the CodeWarrior profiler.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma profile on | off | reset
```

Remarks This pragma applies to Mac OS programming only.

If this pragma is on, the compiler generates code for each function that lets the Metrowerks Profiler collect information on it. For more information, see the *Metrowerks Profiler Manual*.

This pragma corresponds to the **Generate Profiler Calls** option in the 68K Processor settings panel and the Emit Profiler Calls in the PPC Processor settings panel. To check whether this option is on, use `__option (profile)` described in "Checking Options" on page 198.

readonly_strings

Description Controls how to store string literals.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma readonly_strings on | off | reset
```

Remarks This option determines where to stores string constants. If this option is off, the compiler stores string constants in the data section. If this option is on, the compiler stores string constants in the code section.

NOTE: Variables that are not initialized to the address of another object at run time are always placed in the code section (class RO). This includes C/C++ variables declared with the const storage-class modifier.

This pragma corresponds to the **Make Strings ReadOnly** option in the **PPC Processor**, **MIPS**, and **V800 Processor** panels. To check whether this option is on, using `#if __option (readonly_strings)`, see "Checking Options" on page 198.

register_coloring

Description Controls whether the use of register coloring.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma register_coloring on | off | reset
```

Remarks When this pragma is on, the compiler uses a single register to hold the value of more than one variable if those variables are never used in the same statement to improve a program's performance.

TIP: Turn this option off when debugging a program.

This pragma corresponds to the **Register Coloring** option in the **x86 Codegen** panel. To check whether this option is on, use `__option(register_coloring)`, described in "Checking Options" on page 198.

See also "no_register_coloring" on page 143.

require_prototypes

Description Controls whether or not the compiler should expect function prototypes.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma require_prototypes on | off | reset
```

Remarks When the pragma `require_prototypes` is on, the compiler generates an error if you use a function that does not have a prototype.

This pragma helps you prevent errors that happen when you use a function before you define it.

This pragma corresponds to the **Require Function Prototypes** option in the C/C++ Language Panel. To check whether this option is on, use `__option (require_prototypes)`, described in "Checking Options" on page 198.

RTTI

Description Controls the availability of runtime type information.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma RTTI on | off | reset
```

Remarks When this pragma is on, you can use runtime type information (or RTTI) features, such as `dyanamic_cast` and `typeid`. The other RTTI expressions are available even if the **Enable RTTI** option is off. Note that `*type_info::before(const type_info&)` is not yet implemented.

This pragma corresponds to the **Enable RTTI** option in the C/C++ Language Panel. To check whether this option is on, use `__option (RTTI)`, described in "Checking Options" on page 198.

scheduling

Description Specifies the use of instruction scheduling optimization.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma scheduling 601 | 603 | 604 |
                           on | off | reset
```

Remarks

This pragma lets you choose how the compiler rearranges instructions to increase speed. Some instructions, such as a memory load, take more than one processor cycle. By moving an unrelated instruction between the load and the instruction that uses the loaded item, the compiler saves a cycle when executing the program.

For PowerPC, you can use `601`, `603`, or `604`. If you use `on`, the compiler performs 601 scheduling.

However, if you're debugging your code, turn this pragma off. Since it rearranges the instructions produced from your code, the debugger will not be able to match the statements in your source code to the produced instructions.

section

Description

Controls the organization of object code.

Compatibility

This pragma is compatible with the following platform targets:

68K	**PowerPC**	NEC V800	Intel x86	MIPS

Prototype

```
#pragma section [ objecttype | permission ] [iname]
[uname] [data_mode=datamode] [code_mode=codemode]
```

Remarks

This pragma applies to PowerPC embedded programming only.

This sophisticated and powerful pragma lets you arrange compiled object code into predefined sections and sections you define. This topic is organized into these parts:

- Parameters
- Section access permissions
- Predefined sections and default sections
- Forms for #pragma section

- Forcing individual objects into specific sections
- Using #pragma section with #pragma push and #pragma pop

Parameters

The optional *objecttype* parameter specifies where types of object data are stored. It may be one or more of the following values:

- `code_type`—executable object code
- `data_type`—non-constant data of a size greater than the size specified in the small data threshold option in the PowerPC EABI Project settings panel
- `sdata_type`—non-constant data of a size less than or equal to the size specified in the small data threshold option in the PowerPC EABI Project settings panel
- `const_type`—constant data of a size greater than the size specified in the small const data threshold option in the PowerPC EABI Project settings panel
- `sconst_type`—constant data of a size less than or equal to the size specified in the small const data threshold option in the PowerPC EABI Project settings panel
- `all_types`—all data

Specify one or more of these object types without quotes and separated by spaces.

CodeWarrior C/C++ generates some of its own data, such as exception and static initializer objects, which are not affected by `#pragma section`.

NOTE: CodeWarrior C/C++ uses the initial setting of the **Make Strings ReadOnly** option in the **PowerPC EABI Processor** settings panel to classify character strings. If Make Strings ReadOnly is on, character strings are stored in the same section as data of type `const_type`. If Make Strings ReadOnly is off, strings are stored in the same section as data for `data_type`.

The optional *permission* parameter specifies access permission. It may be one or more of these values:

- `R`—read only permission
- `W`—write permission
- `X`—execute permission

For information on access permission, see "Section access permissions" on page 167. Specify one or more of these permissions in any order, without quotes, and no spaces.

The optional *iname* parameter is a quoted name that specifies the name of the section where the compiler stores initialized objects. Variables that are initialized at the time they are defined, functions, and character strings are examples of initialized objects. The iname parameter may be of the form "`.abs.xxxxxxxx`" where *xxxxxxxx* is an 8-digit hexadecimal number specifying the address of the section.

The optional *uname* parameter is a quoted name that specifies the name of the section where the compiler stores uninitialized objects. This parameter is required for sections that have data objects. The uname parameter may be a unique name or it may be the name of any previous *iname* or *uname* section. If the *uname* section is also an *iname* section then uninitialized data will be stored in the same section as initialized objects.

The special *uname* `COMM` specifies that uninitialized data will be stored in the common section. The linker will put all common section data into the "`.bss`" section. When the **Use Common Section** option is on in the **PowerPC EABI Processor** panel, `COMM` is the default *uname* for the "`.data`" section. When the **Use Common Section** option is off, `COMM` is the default *uname* for the "`.bss`" section.

The *uname* parameter may be changed. For example, you may want most uninitialized data to go into the "`.bss`" section while specific variables be stored in the `COMM` section. Listing 6.6 shows an example of specifying that specific uninitialized variables be stored in the `COMM` section.

Listing 6.6 Storing uninitialized data in the COMM section

```
// the Use Common Section option is off
#pragma push // save the current state
#pragma section ".data" "COMM"
int foo;
int bar;
#pragma pop // restore the previous state
```

You may not use any of the object types, data modes, or code modes as the names of sections. Also, you may not use pre-defined section names in the PowerPC EABI for your own section names.

The optional `data_mode=`*datamode* parameter tells the compiler what kind of addressing mode to use for referring to data objects for a section.

The permissible addressing modes for *datamode* are:

- `near_abs`—objects must be within the first 16 bits of RAM
- `far_abs`—objects must be within the first 32 bits of RAM
- `sda_rel`—objects must be within a 32K range of the linker-defined small data base address

  The `sda_rel` addressing mode may only be used with the "`.sdata`", "`.sbss`", "`.sdata2`", "`.sbss2`", "`.EMB.PPC.sdata0`", and "`.EMB.PPC.sbss0`" sections.

The default addressing mode for large data sections is `far_abs`. The default addressing mode for the predefined small data sections is `sda_rel`.

Specify one these addressing modes without quotes.

The optional `code_mode=codemode` parameter tells the compiler what kind of addressing mode to use for referring to executable routines for a section.

The permissible addressing modes for *codemode* are:

- `pc_rel`—routines must be within 24 bits of where it is called from

- `near_abs`—routines must be within the first 24 bits of RAM

The default addressing mode for executable code sections is `pc_rel`.

Specify one these addressing modes without quotes.

NOTE: All sections have a data addressing mode (`data_mode=`*datamode*) and a code addressing mode (`code_mode=`*codemode*). Although the CodeWarrior C/C++ compiler for PowerPC embedded allows you to store executable code in data sections and data in executable code sections, this practice isn't encouraged.

Section access permissions

When you define a section using `#pragma section`, its default access permission is read only. If you change the current section for a particular object type, the compiler adjusts the access permission to allow the storage of objects of that type while continuing to allow objects of previously-allowed object types. Associating `code_type` to a section adds execute permission to that section. Associating `data_type`, `sdata_type`, or `sconst_type` to a section adds write permission to that section.

Occasionally you might create a section without making it the current section for an object type. You might do so to force an object into a section with the `__declspec` keyword. In this case, the compiler will automatically update the access permission for that section to allow the object to be stored in the section, then issue a warning. To avoid such a warning, make sure to give the section the proper access permissions before storing object code or data into it. As with associating an object type to a section, passing a specific permission adds to the permissions that a section already has.

NOTE: Associating an object type with a section sets the appropriate acces permissions for you.

Predefined sections and default sections

The predefined sections set with an object type become the default section for that type. After assigning a non-standard section to an object type, you may rever to the default section with one of the forms in "Forms for #pragma section" on page 168.

The compiler predefines the sections in Listing 6.7.

Listing 6.7 Predefined sections

```
#pragma section code_type ".text" data_mode=far_abs \
 code_mode=pc_rel
#pragma section data_type ".data" ".bss" data_mode=far_abs \
 code_mode=pc_rel
#pragma section const_type ".rodata" ".rodata" data_mode=far_abs \
 code_mode=pc_rel
#pragma section sdata_type ".sdata" ".sbss" data_mode=sda_rel \
 code_mode=pc_rel
#pragma section sconst_type ".sdata2" ".sbss2" data_mode=sda_rel \
 code_mode=pc_rel
#pragma section ".EMB.PPC.sdata0" ".EMB.PPC.sbss0" \
 data_mode=sda_rel code_mode=pc_rel
```

NOTE: The "`.EMB.PPC.sdata0`" and "`.EMB.PPC.sbss0`" sections are predefined as an alternative to the `sdata_type` object type.

Forms for #pragma section

This pragma has these principal forms:

```
#pragma section ".name1"
```

This form simply creates a section called "*.name1*" if it doesn't already exist. With this form, the compiler doesn't store objects in the section without an appropriate, subsequent `#pragma section` statement or an item defined with the `__declspec` keyword. If only one section name is specified, it is considered the name of the

initialized object section, *iname*. If the section is already declared, you may also optionally specify the uninitialized object section, *uname*. If you know that the section is should have read and write permission, use `#pragma section RW ".`*name1*`"` instead, especially if you use the `__declspec` keyword.

`#pragma section` *objecttype* `".`*name2*`"`

With the addition of one or more object types, the compiler stores objects of the types specified in the section "*.name2*". If "*.name2*" doesn't exist, the compiler creates it with the appropriate access permissions. If only one section name is specified, it is considered the name of the initialized object section, *iname*. If the section is already declared, you may also optionally specify the uninitialized object section, *uname*This feature is useful for temporarily circumventing the small data threshold.

`#pragma section` *objecttype*

When there is no *iname* parameter, the compiler resets the section for the object types specified to the default section. For information on predefined sections, see "Predefined sections and default sections" on page 168. Resetting an object type's section doesn't reset its addressing modes. You must do so explicitly.

When declaring or setting sections, you may also add a uninitialized section to a section that didn't have one originally by specifiying a uname parameter. However, you may not change the uninitialized section associated with an initialized section once an uninitialized section has already been associated to it. Remember that an initialized section's corresponding uninitialized section may be the same.

Forcing individual objects into specific sections

You may store a specific object of an object type into a section other than the current section for that type without changing the current section. Use the __declspec keyword with the name of the target section and put it next to the extern declaration or static definition of the item you want to store in the section. Listing 6.8 shows examples.

Listing 6.8 Using __declspec to force objects into specific sections

```
__declspec(".data") extern int myVar;
#pragma section "constants"
__declspec("constants") const int myvar = 0x12345678;
```

Using #pragma section with #pragma push and #pragma pop

This pragma may be used with `#pragma push` and `#pragma pop` to ease complex or frequent changes to sections settings. See Listing 6.6 for an example. Note that `#pragma pop` doesn't restore any changes to the access permissions of sections that exists before or after the corresponding `#pragma push`.

segment

Description Specifies the code segment where subsequent object code should be stored.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma segment` *name*

Remarks This pragma applies to Mac OS programming only.

This pragma places all the functions that follow into the code segment named *name*. For more on function-level segmentation, consult the *Targeting* manual for the platform you're developing for.

Generally, the PowerPC compilers ignore this directive since PowerPC applications do not have code segments. However, if you choose **by #pragma segment** from the **Code Sorting** pop-up menu in the **PPC PEF** settings panel, the PowerPC compilers group functions in the same segment together. For more information, consult the *Targeting* manual for the platform you're developing for.

This pragma does not correspond to an option in any settings panel.

side_effects

Description Controls the use of pointer aliases.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma side_effects on | off | reset
```

Remarks If your program does not use pointer aliases, turn off this pragma to make your program smaller and faster. If your program does use pointer aliases, turn on this pragma to avoid incorrect code. A pointer alias looks like this:

```
int a, *p;
p = &a;   // *p is an alias for a.
```

To understand why pointer aliases are so important, remember that the compiler needs to load a variable into a register before performing arithmetic on it. So, in the example below, the compiler loads `a` into a register before the first addition. If `*p` is an alias for `a`, the compiler needs to load `a` into a register again before the second addition, since changing `*p` also changed `a`. If `*p` is not an alias for `a`, the compiler doesn't need to load `a` into a register again, since changing `*p` does not change `a`.

```
x = a + 1;
*p = 0;      // If *p is an alias for a,
y = a + 2;   //  this changes a.
```

NOTE: The PowerPC compilers ignore this pragma and always assume that a program may contain pointer aliases.

This pragma does not correspond to an option in any settings panel. To check whether this pragma is on, use `__option`

`(side_effects)`, described in "Checking Options" on page 198. By default, this pragma is on.

simple_prepdump

Description Controls the suppression of comments in preprocessor dumps.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma simple_prepdump on | off | reset
```

Remarks By default, the preprocessor adds comments about the current include file being processed in its output. These comments can be disabled by turning this pragma on.

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option (simple_prepdump)`. See on "Checking Options" on page 198. By default, this pragma is off.

SOMCallOptimization

Description Controls the error checking used for making calls to SOM objects.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma SOMCallOptimization on | off | reset
```

Remarks This pragma is only available for Mac OS using C++ code.

The PowerPC compiler uses an optimized error check that is smaller but slightly slower.

This pragma is ignored if the `direct_to_SOM` pragma, described in "direct_to_som" on page 111, is off.

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option (SOMCall-Optimization)`. See on "Checking Options" on page 198. By default, this pragma is off.

SOMCallStyle

Description Specifies the convention used to call SOM objects.

Compatibility This pragma is compatible with the following platform targets:

68K	**PowerPC**	NEC V800	Intel x86	MIPS

Prototype

```
#pragma SOMCallStyle OIDL | IDL
```

Remarks This pragma is only available for Mac OS using C++ code.

The `SOMCallStyle` pragma chooses between two SOM call styles:

- OIDL, an older style that does not support DSOM
- IDL, a newer style that does support SOM.

If a class uses the IDL style, its methods must have an Environment pointer as the first parameter. Note that the SOMClass and SOMObject classes use OIDL, so if you override a method from one of them, you should not include the Environment pointer.

This pragma is ignored if the `direct_to_SOM` pragma, described in *Targeting Mac OS*, is off.

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option (SOMCheckEnvironment)`. See "Checking Options" on page 198. By default, this pragma is set to `IDL`.

SOMCheckEnvironment

Description Controls whether or not to perform SOM environment checking.

Compatibility This pragma is compatible with the following platform targets:

68K	**PowerPC**	NEC V800	Intel x86	MIPS

Prototype `#pragma SOMCheckEnvironment on | off | reset`

Remarks This pragma is only available for Mac OS using C++ code.

When the pragma `SOMCheckEnvironment` is on, the compiler performs automatic SOM environment checking. It transforms every IDL method call and new allocation into an expression which also calls an error-checking function. You must define separate error-checking functions for method calls and allocations. For more information on how to write these functions, see *Targeting Mac OS*.

For example, the compiler transforms this IDL method call:

```
SOMobj->func(&env, arg1, arg2) ;
```

into something that is equivalent to this:

```
( temp=SOMobj->func(&env, arg1, arg2),
  __som_check_ev(&env), temp ) ;
```

First, the compiler calls the method and stores the result in a temporary variable. Then it checks the environment pointer. Finally, it returns the method's result.

And, the compiler transforms this new allocation:

```
new SOMclass;
```

into something that is equivalent to this:

```
( temp=new SOMclass, __som_check_new(temp),
  temp);
```

First, the compiler creates the object and stores it in a temporary variable. Then it checks the object and returns it.

The PowerPC compiler uses an optimized error check that is smaller but slightly slower than the one given above. To use the error check show above in PowerPC code, use the pragma `SOMCall-Optimization`, described in "SOMCallOptimization" on page 172.

This pragma is ignored if the `direct_to_SOM` pragma, described in *Targeting Mac OS* is off.

This pragma corresponds to the **Direct to SOM** menu in the C/C++ Language Panel. Selecting **On with Environment Checks** from that menu is like setting this pragma to `on`. Selecting anything else from that menu is like setting this pragma to `off`. To check whether this option is on, use `__option (RTTI)`, described in "Checking Options" on page 198. By default, this pragma is on.

SOMClassVersion

Description Specifies a SOM class's version.

Compatibility This pragma is compatible with the following platform targets:

68K	**PowerPC**	NEC V800	Intel x86	MIPS

Prototype

```
#pragma SOMClassVersion(class,majorVer,minorVer)
```

Remarks This pragma is only available for Mac OS using C++ code.

SOM uses the class's version number to make sure the class is compatible with other software you're using. If you don't declare the version numbers, SOM assumes zeroes. The version numbers must be positive or zero.

When you define the class, the program passes its version number to the SOM kernel in the class's metadata. When you instantiate an object of the class, the program passes the version to the runtime kernel, which checks to make sure the class is compatible with the running software.

This pragma is ignored if the `direct_to_SOM` pragma, described in *Targeting Mac OS*, is off.

This pragma does not correspond to an option in any settings panel.

SOMMetaClass

Description Specifies a SOM class's metaclass.

Compatibility This pragma is compatible with the following platform targets:

68K	**PowerPC**	NEC V800	Intel x86	MIPS

Prototype

```
#pragma SOMMetaClass (class, metaclass)
```

Remarks This pragma is only available for Mac OS using C++ code.

A metaclass is a special kind of SOM class that defines the implementation of other SOM classes. All SOM classes have a metaclass, including metaclasses themselves. By default, the metaclass for a SOM class is SOMClass. If you want to use another metaclass, use the `SOMMetaClass` pragma:

The metaclass must be a descendant of SOMClass. Also, a class cannot be its own metaclass. That is, *class* and *metaclass* must name different classes.

This pragma is ignored if the `direct_to_SOM` pragma, described in *Targeting Mac OS*, is off.

This pragma does not correspond to an option in any settings panel.

SOMReleaseOrder

Description Specifies the order in which a SOM class's member functions are released.

Compatibility This pragma is compatible with the following platform targets:

68K	**PowerPC**	NEC V800	Intel x86	MIPS

Prototype `#pragma SOMRelaseOrder(`*func1*`, `*func2*`,... `*funcN*`)`

Remarks This pragma is only available for Mac OS using C++ code.

A SOM class must specify the release order of its member functions. As a convenience for when you're first developing the class, Metrowerks C++ lets you leave out the `SOMReleaseOrder` pragma and assumes the release order is the same as the order in which the functions appear in the class declaration. However, when you release a version of the class, use the pragma, since you'll need to modify its list in later versions of the class.

You must specify every SOM method that the class introduces. Do not specify inline member functions that are virtual, since they're not considered to be SOM methods. Don't specify overridden functions.

If you remove a function from a later version of the class, leave its name in the release order list. If you add a function, place it at the end of the list. If you move a function up in the class hierarchy, leave it in the original list and add it to the list for the new class.

This pragma is ignored if the `direct_to_SOM` pragma, described in *Targeting Mac OS*, is off.

This pragma does not correspond to an option in any settings panel.

stack_cleanup

Description Controls when the compiler generates code to clean up the stack.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma stack_cleanup on | off | reset`

Remarks Turning this option on will disable the deferred stack cleanup after function calls, forcing the compiler to remove arguments from the stack after every function call. Although this option slows down ex-

ecution, it reduces stack usage, making it less likely the stack will intrude on other parts of the program.

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option (stack_cleanup)`, described in "Checking Options" on page 198. By default, this pragma is off.

static_inlines

Description Controls how many instances of inline functions that the compiler generates.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma static_inlines on | off | reset
```

Remarks The pragma `static_inlines` determines what the compiler does if it cannot inline a call to a function declared `inline` and must create a compiled version of the function. If the pragma is off, the compiler creates one compiled version for the whole project. If the pragma is on, the compiler creates a different compiled version for each file that needs a compiled version.

This pragma is available only so that the compiler can pass certain validation suites. Generally, you'll want to leave this pragma off to make your code smaller without any loss of speed.

This pragma does not correspond to an option in any settings panel. To check whether this option is on, use `__option (static_inlines)`, described in "Checking Options" on page 198. By default, this pragma is off.

suppress_init_code

Description Controls the suppression of static initialization object code.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma suppress_init_code on | off | reset
```

Remarks When this pragma is on, the compiler doesn't generate any code for static data initialization such as C++ constructors. By default this pragma is off.

WARNING! Using this pragma without being aware of its consequences can produce erratic or unpredictable behavior in your program.

sym

Description Controls the generation of debugger symbol information.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma sym on | off | reset
```

Remarks The compiler pays attention to this pragma only if you turn on the debug marker for a file in the IDE's project window. If this pragma is off, the compiler does not put debugging information into this source file's debugger symbol file (SYM or DWARF) for the functions that follow. If this pragma is on, the compiler does generate debugging information.

Note that the compiler always generates a debugger symbol file for a source file that has a debug diamond next to it in the project window. This pragma changes only which functions have information in that symbol file.

To check whether this option is on, use `__option (sym)`, described in "Checking Options" on page 198. By default, this pragma is on.

syspath_once

Description Controls how include files are treated.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma syspath_once on | off | reset
```

Remarks Files referred to in `#include <>` and `#include ""` directives are treated as distinct files if this option is selected, even if they refer to the same file.

toc_data

Description Controls how static variables are stored.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma toc_data on | off | reset
```

Remarks This pragma applies to Mac OS CFM programming only.

If the `toc_data` pragma is on, the compiler makes your code smaller and faster by storing static variables that are 4 bytes or smaller directly in the TOC, instead of allocating space for them elsewhere and storing pointers to them in the TOC. Turn this pragma off only if your code expects the TOC to contain pointers to data.

This pragma corresponds to the **Store Static Data in TOC** option in the PPC Processor settings panel. To check whether this option is

on, use `__option (toc_data)`, described in "Checking Options" on page 198.

trigraphs

Description Controls the use ANSI/ISO trigraph sequences.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma trigraphs on | off | reset
```

Remarks If you're writing code that must follow the ANSI standard strictly, turn on the pragma `trigraphs` in the C/C++ Language Panel. Many common Macintosh character constants look like trigraph sequences, and this pragma lets you use them without including escape characters. Be careful when you initialize strings or multi-character constants that contain question marks. For example:

```
char c = '????';     // ERROR: Trigraph sequence expands to '??^
char d = '\?\?\?\?'; // OK
```

This pragma corresponds to the **Expand Trigraphs** option in the C/C++ Language Panel. To check whether this option is on, use `__option (trigraphs)`, described in "Checking Options" on page 198.

traceback

Description Controls the generation of AIX-format traceback tables for debugging.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma traceback on | off | reset
```

Remarks

This pragma helps other people debug your application or shared library if you do not distribute the source code. If this option is on, the compiler generates an AIX-format traceback table for each function, which are placed in the executable code. Both the Metrowerks and Apple debuggers can use traceback tables.

This pragma corresponds to the **Emit Traceback Tables** option in the PPC Linker settings panel. To check whether this option is on, use the `__option (traceback)`, described in "Checking Options" on page 198. By default, this option is off.

unsigned_char

Description

Controls whether or not declarations of type `char` are treated as `unsigned char`.

Compatibility

This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma unsigned_char on | off | reset
```

Remarks

When the `unsigned_char` pragma is on, the C/C++ compiler treats a `char` declaration as if it were an `unsigned char` declaration.

NOTE: If you turn this pragma on, your code may not be compatible with libraries that were compiled with it turned off. In particular, your code may not work with the ANSI libraries included with CodeWarrior.

This pragma corresponds to the **Use unsigned chars** option in the C/C++ Language Panel. To check whether this option is on, use `__option (unsigned_char)`, described in "Checking Options" on page 198. By default, this option is off.

unused

Description Controls the suppression of warnings for variables and parameters that aren't referenced in a function.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma unused ( var_name [, var_name ]... )`

Remarks This pragma suppresses the compile time warnings for the unused variables and parameters specified in its argument list. You can use this pragma only within a function body, and the listed variables must be within the function's scope. You cannot use this pragma with functions defined within a class definition or with template functions. For example:

```
#pragma warn_unusedvar on
#pragma warn_unusedarg on

static void ff(int a)
{
  int b;
#pragma unused(a,b)  // Compiler won't complain
                     // that a and b are unused
  // . . .
}
```

This pragma does not correspond to an option in any settings panel.

use_fp_instructions

Description Controls the generation of NEC V800 floating point instructions.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma use_fp_instructions on|off|reset`

Remarks This option corresponds to the option **Use V810 Floating-Point Instructions**, which is part of the **NEC V800 Processor** panel. To check whether this option is on, use `__option (use_fp_instructions)`, described in "Checking Options" on page 198.

use_frame

Description Controls the use of the BP register for stack frames.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	**Intel x86**	MIPS

Prototype `#pragma use_frame on|off|reset`

Remarks When this option is on the compiler uses the BP register to point to the start of the stack frame.

To check whether this option is on, use `__option (use_frame)`, described in "Checking Options" on page 198.

use_mask_registers

Description Controls the use of the NEC V800 r20 and r21 registers.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	**NEC V800**	Intel x86	MIPS

Prototype `#pragma use_mask_registers on|off|reset`

Remarks This option corresponds to the option **Use r20 and r21 as Mask Registers**, which is part of the **NEC V800 Processor** panel. To check whether this option is on, use `__option`

`(use_mask_registers)`, described in "Checking Options" on page 198.

warn_emptydecl

Description Controls the recognition of declarations without variables.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma warn_emptydecl on | off | reset
```

Remarks If the pragma `warn_emptydecl` is on, the compiler displays a warning when it encounters a declaration with no variables. For example:

```
int ;      // WARNING
int i;     // OK
```

This pragma corresponds to the **Empty Declarations** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warn_emptydecl)`, described in "Checking Options" on page 198.

warning_errors

Description Controls whether warnings are treated as errors or not.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma warning_errors on | off | reset
```

Remarks When the pragma `warning_errors` is on, the compiler treats all warnings as though they were errors. It will not compile a file until all warnings are resolved.

This pragma corresponds to the **Treat All Warnings as Errors** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warning_errors)`, described in "Checking Options" on page 198.

warn_extracomma

Description Controls the recognition of superfluous commas.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma warn_extracomma on | off | reset
```

Remarks If the pragma `warn_extracomma` is on, the compiler generates a warning when it encounters an extra comma. For example, this statement is legal in C, but it causes a warning when this pragma is on:

```
int a[] = { 1, 2, 3, 4, }; // ^ WARNING: Extra comma after 4
```

This pragma corresponds to the **Treat All Warnings as Errors** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warn_extracomma)`, described in "Checking Options" on page 198.

warn_hidevirtual

Description Controls the recognition of a non-virtual member function that hides a virtual function in a superclass.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma warn_hidevirtual on|off|reset`

Remarks If the pragma `warn_hidevirtual` is on, the compiler generates a warning if you declare a non-virtual member function that hides a virtual function in a superclass. One function hides another if it has the same name but a different argument types. For example:

```
class A {
  public:
    virtual void f(int);
    virtual void g(int);
};

class B: public A {
  public:
    void f(char);         // WARNING: Hides A::f(int)
    virtual void g(int); // OK: Overrides A::g(int)
};
```

This pragma corresponds to the **Hidden virtual functions** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warn_hidevirtual)`. See "Checking Options" on page 198. By default, this option is off.

warn_illpragma

Description Controls the recognition of illegal pragma directives.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma warn_illpragma on | off | reset`

Remarks If the pragma `warn_illpragma` is on, the compiler displays a warning when it encounters an illegal pragma. For example, these pragma statements generate warnings:

```
#pragma near_data off   // WARNING: near_data is not a pragma.
#pragma far_data select // WARNING: select is not defined
#pragma far_data on     // OK
```

This pragma corresponds to the **Illegal Pragmas** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warn_illpragma)`, described in "Checking Options" on page 198.

warn_implicitconv

Description Controls the issuing of warnings for implicit arithmetic conversions.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma warn_implicitconv on | off | reset
```

Remarks The compiler will print a warning for implicit arithmetic conversions when the source value may not be representable by the destination type. See Listing 6.9 for an example.

Listing 6.9 Example of implicit arithmetic conversion

```
#pragma warn_implicitconv on

char foo(int a)
{
  return a+1; // Warning : implicit arithmetic conversion ...
              // ... from 'int' to 'char'
}
```

This pragma corresponds to the **Implicit Arithmetic Conversion** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warn_implicitconv)`, described in "Checking Options" on page 198.

warn_notinlined

Description Controls the issuing of warnings for functions the compiler isn't able to inline.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma warn_notinlined on | off | reset
```

Remarks The compiler will issue a warning if it is not able to inline a function. This pragma corresponds to the **Non-Inlined Functions** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warn_notinlined)`, described in "Checking Options" on page 198.

warn_possunwant

Description Controls the recognition of possible unintentional logical errors.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma warn_possunwant on | off | reset
```

Remarks If the pragma `warn_possunwant` is on, the compiler checks for some common typographical mistakes that are legal C and C++ but that may have unwanted side effects, such as putting in unintended semicolons or confusing = and ==. The compiler generates a warning if it encounters one of these:

- An assignment in a logical expression or the condition in an `if`, `while`, or `for` expression. This check is useful if you frequently use = when you meant to use ==. For example:

```
if (a=b) f();       // WARNING: a=b is an assignment

if ((a=b)!=0) f(); // OK: (a=b)!=0 is a comparison
```

```
if (a==b) f();      // OK: (a==b) is a comparison
```

- An equal comparison in a statement that contains a single expression. This check is useful if you frequently use == when you meant to use =. For example:

```
a == 0;     // WARNING: This is a comparison.
a = 0;      // OK: This is an assignment
```

- A semicolon (;) directly after a `while`, `if`, or `for` statement. For example, the statement generates an error and is probably an unintended infinite loop:

```
while (i++); // WARNING: Unintended infinite loop
```

If you intended to create an infinite loop, put white space or a comment between the `while` statement and the semicolon, and earn the admiration of all the folks who use your code. For example, these statements do not generate errors:

```
while (i++) ;     // OK: White space separation
while (i++)       /* OK: Comment separation */ ;
```

This pragma corresponds to the **Possible Errors** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warn_possunwant)`, described in "Checking Options" on page 198.

warn_structclass

Description Controls the issuing of warnings for possibly unintended mixing of `class` and `struct` keywords.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma warn_structclass on | off | reset`

Remarks If the warn_structclass pragma is on, the compiler issues a warning if the `class` and `struct` keywords are used in the definition and declaration of the same identifier.

```
class X;
struct X { int a; }; // warning
```

This pragma corresponds to the **Inconsistent use of 'class' and 'struct' Keywords** option in the **C/C++ Warnings** settings panel. To check whether this option is on, use `__option (warn_structclass)`, described in "Checking Options" on page 198.

warn_unusedarg

Description Controls the recognition of unreferenced arguments.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype `#pragma warn_unusedarg on | off | reset`

Remarks If the pragma `warn_unusedarg` is on, the compiler generates a warning when it encounters an argument you declare but do not use. This check helps you find misspelled argument names and arguments you have written out of your program.

```
void foo(int temp, int errer)
{
  error = do_something();  // ERROR: Error is undefined
} // WARNING: temp and error are unused.
```

This pragma corresponds to the **Unused Arguments** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warn_unusedarg)`, described in "Checking Options" on page 198.

warn_unusedvar

Description Controls the recognition of unreferenced variables.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma warn_unusedvar on | off | reset
```

Remarks If the pragma `warn_unusedvar` is on, the compiler generates a warning when it encounters a variable you declare but do not use. This check helps you find misspelled variable names and variables you have written out of your program. For example:

```
void foo(void)
{
  int temp, errer;
  error = do_something();  // ERROR: error is undefined
}    // WARNING: temp and error are unused.
```

This pragma corresponds to the **Unused Variables** option in the C/C++ Warnings settings panel. To check whether this option is on, use `__option (warn_unusedvar)`, described in "Checking Options" on page 198.

warning

Description Available for compatibility only.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma warning(warning_specifier : warning_number_list)
```

Remarks This pragma applies to x86 programming only.

Ignored. Included for compatibility with Microsoft. The *warning_number_list* is a list of warning numbers separated by spaces, and *warning_specifier* is one of the following:

- `once`
- `default`
- `1`
- `2`
- `3`
- `4`
- `disable`
- `error`

wchar_type

Description Controls the size and format of the `wchar_t` type.

Compatibility This pragma is compatible with the following platform targets:

68K	PowerPC	NEC V800	Intel x86	MIPS

Prototype

```
#pragma wchar_type on | off | reset
```

Remarks If the pragma `wchar_type` is on, `wchar_t` is treated as a built-in type, implemented as an unsigned 16-bit integral type. If the pragma is off, `wchar_t` and characters in string literals are treated as `unsigned short`.

This pragma corresponds to the **Enable wchar_t Support** option in the C/C++ Language settings panel. To check this option, use `__option (wchar_type)`, described in "Checking Options" on page 198.

Predefined Symbols

Metrowerks C and C++ define several preprocessor symbols that give you information about the compile-time environment. Note

that these symbols are evaluated at compile time and not at run time. The topics in this section are:

- ANSI Predefined Symbols
- Metrowerks Predefined Symbols

ANSI Predefined Symbols

The table below lists the symbols that the ANSI C standard requires.

Table 6.7 ANSI predefined symbols

This macro…	is…
`__DATE__`	The date at which the file is compiled; for example, `"Jul 14, 1995"`.
`__FILE__`	The name of the file being compiled; for example `"prog.c"`.
`__LINE__`	The line number of the line being compiled. This is the number before including any header files.
`__TIME__`	The time at which the file is compiled in 24-hour format; for example, `"13:01:45"`.
`__STDC__`	Defined as `1` if compiling C source code. This macro lets you know that Metrowerks C implements the ANSI C standard. This macro is undefined when compiling C++ source code.

Listing 6.10 shows a small program that uses the ANSI predefined symbols.

Listing 6.10 Using ANSI's Predefined Symbols

```
#include <stdio.h>

void main(void)
{
  printf("Hello World!\n");
```

```
  printf("%s, %s\n", __DATE__, __TIME__);
  printf("%s, line: %d\n",  __FILE__, __LINE__);
}
```

The program prints something like the following:

```
Hello World!
Oct 31 1995, 18:23:50
main.ANSI.c, line: 10
```

Metrowerks Predefined Symbols

The table below lists additional symbols that Metrowerks C and C++ provides.

Table 6.8 Predefined symbols for Metrowerks

This macro…	is…
`__A5__`	defined as `1` if data is A5-relative, `0` if data is A4 relative. This symbol is defined for 68K compilers. It is undefined for other target platforms.
`__cplusplus`	defined if you're compiling this file as a C++ file, undefined if you're compiling this file as a C file.
`__embedded_cplusplus`	defined as 1 if Embedded C++ is activated. This symbol is undefined if Embedded C++ is not activated. See
`__fourbyteints__`	defined as `1`, if you turn on the **4-byte Ints** option in the 68K Processor settings panel. `0`, if you turn off that option. This symbol is defined for 68K compilers. It is undefined for other platforms.

This macro…	is…
`__IEEEdoubles__`	defined as `1` if you turn on the **8-Byte Doubles** option in the 68K Processor settings panel. `0`, if you turn off that option. This symbol is defined for 68K compilers. It is undefined for all other target platforms.
`__INTEL__`	defined as `1` if you're compiling this code with the x86 compiler and undefined for all other target platforms.
`__MC68K__`	defined as `1` if you're compiling this code with the 68K compiler and undefined for all other target platforms.
`__MC68020__`	defined as `1` if you turn on the **68020 Codegen** option in the Processor settings panel, `0` if you turn that option off. This symbol is defined for 68K compilers. It is undefined for all other target platforms.
`__MC68881__`	defined as `1` if you turn on the **68881 Codegen** option in the 68K Processor settings panel, `0` if you turn that option off. This symbol is defined for 68K compilers and undefined for all other target platforms.
`__MIPS__`	defined as `1` for MIPS compilers and undefined for other target platforms.
`__MIPS_ISA2__`	defined as `1` if the compiler's target platform is MIPS and you select the **ISA II** checkbox in the **MIPS Processor** settings panel. Undefined if you deselect the checkbox. It is always undefined for other target platforms.

This macro…	is…
`__MIPS_ISA3__`	defined as 1 if the compiler's target platform is MIPS and you select the **ISA III** checkbox in the **MIPS Processor** settings panel. Undefined if you deselect the checkbox. It is always undefined for other target platforms.
`__MIPS_ISA4__`	defined as 1 if the compiler's target platform is MIPS and you select the **ISA IV** checkbox in the **MIPS Processor** settings panel. Undefined if you deselect the checkbox. It is always undefined for other target platforms
`__MWBROWSER__`	defined as `1` if the CodeWarrior browser is parsing your code. `0`, otherwise.
`__MWERKS__`	defined as the version number of the Metrowerks C/C++ compiler, if you're using CodeWarrior CW7 or later. For example, in Metrowerks C/C++ version 2.2 __MWERKS__ would be `0x2200`. This macro is defined as 1 if the compiler was issued before CodeWarrior CW7.
`__profile__`	`1`, if you turn on the **Generate Profiler Calls** option in the Processor settings panel. `0`, if you turn that option off.
`__powerc`, `powerc`, `__POWERPC__`	`1`, if you're compiling this code with the PowerPC compiler. `0`, otherwise.
`macintosh`	`1` if you're compiling this code with the 68K or PowerPC Macintosh compiler. `0` otherwise.

Checking Options

The preprocessor function `__option()` lets you test the setting of many pragmas and options that control the C/C++ compiler and code generation. You typically modify these settings using various panels in the Project Settings dialog box.

The syntax for this preprocessor function is:

```
__option(option-name)
```

If the specified option is on, `__option ()` returns 1; otherwise, it returns 0.

This function is useful when you want one source file to contain code that uses different option settings. The example below shows how to compile one series of lines if you're compiling for machines with the MC68881 floating-point unit and another series if you're compiling for machines without out:

```
#if __option (code68881)  // Code for 68K chip with FPU
#else
  // Code for any 68K processor
#endif
```

The table below lists all the option names you can use in the preprocessor function `__option()`.

This argument...	Corresponds to the…
`a6frames`	**Generate A6 Stack Frames** option in the 68K Linker settings panel and pragma `a6frames`.
`align_array_members`	Pragma `align_array_members`.
`always_inline`	Pragma `always_inline`.
`ANSI_strict`	**ANSI Strict** option in the C/C++ Language Panel and pragma `ANSI_strict`.
`arg_dep_lookup`	Pragma `arg_dep_lookup`.

This argument...	Corresponds to the…
`ARM_conform`	**ARM Conformance** option in the C/C++ Language Panel and pragma `ARM_conform`.
`auto_inline`	**Auto-Inline** option of the **Inlining** menu in the C/C++ Language Panel and pragma `auto_inline`.
`bool`	Enable **C++ bool/true/false** option in the C/C++ Language Panel and pragma `bool`.
`check_header_flags`	Pragma `check_header_flags`.
`code68020`	**68020 Codegen** option in the 68K Processor settings panel and pragma `code68020`.
`code68881`	**68881 Codegen** option in the 68K Processor settings panel and pragma `code68881`.
`cplusplus`	Whether the compiler is compiling this file as a C++ file. Related to the **Activate C++ Compiler** option in the C/C++ Language Panel, the pragma `cplusplus`, and the macro `cplusplus`
`cpp_extensions`	Pragma `cpp_extensions`
`d0_pointers`	Pragmas `pointers_in_D0` and `pointers_in_A0`.
`def_inherited`	Pragma `def_inherited`.
`defer_codegen`	Pragma `defer_codegen`.
`direct_destruction`	**Enable Exception Handling** option in the C/C++ Language Panel and pragma `direct_destruction`.
`direct_to_SOM`	**Direct to SOM** menu in the C/C++ Language Panel and pragma `direct_to_SOM`
`disable_registers`	Pragma `disable_registers`.
`dollar_identifiers`	Pragma `dollar_identifiers`.
`dont_inline`	**Don't Inline** option in the C/C++ Language Panel and pragma `dont_inline`.

This argument...	Corresponds to the...
`dont_reuse_strings`	**Don't Reuse Strings** option in the C/C++ Language Panel and pragma `dont_reuse_strings`.
`ecplusplus`	Pragma `ecplusplus`
`EIPC_EIPSW`	Pragma `EIPC_EIPSW`
`enumsalwaysint`	**Enums Always Int** option in the C/C++ Language Panel and pragma `enumsalwaysint`
`exceptions`	**Enable C++ Exceptions** option in the C/C++ Language Panel and pragma `exceptions`
`export`	Pragma `export`.
`extended_errorcheck`	**Extended Error Checking** option in the C/C++ Warnings settings panel and pragma `extended_errorcheck`.
`far_data`	**Far Data** option in the 68K Processor settings panel and pragma `far_data`.
`far_strings`	**Far String Constants** option in the 68K Processor settings panel and pragma `far_strings`.
`far_vtables`	Far Method Tables in the 68K Processor settings panel and pragma `far_vtables`.
`faster_pch_gen`	Pragma `faster_pch_gen`.
`force_active`	Pragma `force_active`.
`fourbyteints`	**4-Byte Ints** option in the 68K Processor settings panel and pragma `fourbyteints`.
`fp_contract`	**Use FMADD & FMSUB** option in the PPC Processor settings panel and pragma `fp_contract`.
`global_optimizer`	**Global Optimization** option in the PPC Processor settings panel and pragma `global_optimizer`.
`IEEEdoubles`	**8-Byte Doubles** option in the 68K Processor settings panel and pragma `IEEEdoubles`.
`ignore_oldstyle`	Pragma `ignore_oldstyle`.

This argument...	Corresponds to the…
`import`	Pragma `import`.
`inline_intrinsics`	Pragma `inline_intrinsics`.
`internal`	Pragma `internal`.
`interrupt`	Pragma `interrupt`.
`k63d`	**K6 3D Favored** option in the **Extended Instruction Set** menu of the **x86 CodeGen** settings panel and pragma `k63d`.
`k63d_calls`	**MMX + K6 3D** option in the **Extended Instruction Set** menu of the **x86 CodeGen** settings panel and pragma `k63d_calls`.
`lib_export`	Pragma `lib_export`.
`little_endian`	No option. It is 1 if you're compiling for a little endian target (such as x86) and 0 if you're compiling for a big endian target (such as Mac OS).
`longlong`	Pragma `longlong`.
`longlong_enums`	Pragma `longlong_enums`.
`macsbug`	**MacsBug Symbols** option in the 68K Linker settings panel and pragma `macsbug`.
`microsoft_exceptions`	Pragma `microsoft_exceptions`.
`microsoft_RTTI`	Pragma `microsoft_RTTI`.
`mmx`	**MMX** option in the **Extended Instruction Set** menu of the **x86 CodeGen** settings panel and pragma `mmx`.
`mmx_call`	Pragma `mmx_call`.
`mpwc`	**MPW C Calling Conventions** option in the 68K Processor settings panel and pragma `mpwc`.
`mpwc_newline`	**Map Newlines to CR** option in the C/C++ Language Panel and pragma `mpwc_newline`.

This argument...	Corresponds to the…
`mpwc_relax`	**Relaxed Pointer Type Rules** option in the C/C++ Language Panel and pragma `mpwc_relax`.
`no_register_coloring`	**Global Register Allocation** option in the 68K Processor settings panel and pragma `no_register_coloring`.
`oldstyle_symbols`	**MacsBug Symbols** option in the 68K Linker settings panel and pragma `oldstyle_symbols`
`only_std_keywords`	**ANSI Keywords Only** option in the C/C++ Language Panel and pragma `only_std_keywords`.
`opt_common_subs`	Pragma `opt_common_subs`.
`opt_dead_assignments`	Pragma `opt_dead_assignments`.
`opt_dead_code`	Pragma `opt_dead_code`.
`opt_lifetimes`	Pragma `opt_lifetimes`.
`opt_loop_invariants`	Pragma `opt_loop_invariants`.
`opt_propagation`	Pragma `opt_propagation`.
`opt_strength_reduction`	Pragma `opt_strength_reduction`.
`opt_unroll_loops`	Pragma `opt_unroll_loops`.
`opt_vectorize_loops`	Pragma `opt_vectorize_loops`.
`pool_data`	**Pool Data** option in the PPC Processor (for embedded PowerPC programming only) and pragma `pool_data`
`pool_strings`	**Pool Strings** option in the C/C++ Language Panel and pragma `pool_strings`
`precompile`	Whether the file is being pre-compiled.
`preprocess`	Whether the file is being pre-processed
`profile`	**Generate Profiler Calls** option in the 68K Processor settings panel, **Emit Profiler Calls** option in the PPC Processor settings panel, and pragma `profile`.

This argument...	Corresponds to the…
`readonly_strings`	**Make String Literals Readonly** option in the PPC Processor settings panel and pragma `readonly_strings`.
`register_coloring`	Pragma `register_coloring`.
`require_prototypes`	**Require Function Prototypes** option in the C/C++ Language Panel and pragma `require_prototypes`.
`RTTI`	**Enable RTTI** option in the C/C++ Language Panel and pragma `RTTI`.
`side_effects`	Pragma `side_effects`.
`simple_prepdump`	Pragma `simple_prepdump`
`SOMCallOptimization`	Pragma `SOMCallOptimization`
`SOMCheckEnvironment`	**Direct to SOM** menu in the C/C++ Language Panel and pragma `SOMCheckEnvironment`
`static_inlines`	Pragma `static_inlines`
`stack_cleanup`	Pragma `stack_cleanup`.
`suppress_init_code`	Pragma `suppress_init_code`.
`sym`	Marker in the project window debug column and pragma `sym`
`syspath_once`	Pragma `syspath_once`.
`toc_data`	**Store Static Data in TOC** option in the PPC Processor settings panel and pragma `toc_data`
`traceback`	Pragma `traceback`.
`trigraphs`	**Expand Trigraphs** option in the C/C++ Language Panel and pragma `trigraphs`.
`unsigned_char`	**Use Unsigned Chars** option in the C/C++ Language Panel and pragma `unsigned_char`.

This argument...	Corresponds to the…
`use_fp_instructions`	**Use V810 Floating-Point Instructions**, which is part of the **NEC V800 Processor** and pragma `use_fp_instructions`.
`use_frame`	Pragma `use_frame`.
`use_mask_registers`	**Use r20 and r21 as Mask Registers**, which is part of the **NEC V800 Processor** and pragma `use_mask_registers`.
`warn_emptydecl`	**Empty Declarations** option in the C/C++ Warnings settings panel and pragma `warn_emptydecl`.
`warn_extracomma`	**Extra Commas** option in the C/C++ Warnings settings panel and pragma `warn_extracomma`.
`warn_hidevirtual`	**Hidden virtual functions** option in the C/++ Warnings settings panel and pragma `warn_hidevirtual`.
`warn_illpragma`	**Illegal Pragmas** option in the C/C++ Warnings settings panel and pragma `warn_illpragma`.
`warn_implicitconv`	**Implicit Arithmetic Conversions** option in the C/C++ Warnings settings panel and pragma `warn_implicitconv`.
`warn_notinlined`	**Non-Inlined Functions** option in the C/C++ Warnings settings panel and pragma `warn_notinlined`.
`warn_possunwant`	**Possible Errors** option in the C/C++ Warnings settings panel and pragma `warn_possunwant`.
`warn_structclass`	**Inconsistent Use of 'class' and 'struct' Keywords** option in the C/C++ Warnings settings panel and pragma `warn_structclass`.
`warn_unusedarg`	**Unused Arguments** option in the C/C++ Warnings settings panel and pragma `warn_unusedarg`.
`warn_unusedvar`	**Unused Variables** option in the C/C++ Warnings settings panel and pragma `warn_unusedvar`.

This argument...	Corresponds to the…
`warning_errors`	**Treat Warnings As Errors** option in the C/C++ Warnings settings panel and pragma `warning_errors`.
`wchar_type`	**Enable wchar_t Support** option in the C/C++ Language settings panel and pragma `wchar_type`.

Index

Symbols

#, and macros 38
#else 39
#endif 39
#pragma statements
    illegal 20
#pragma statements 88
=
    accidental 21
    operator 57
?: conditional operator 63
__A5__ 195
__builtin_align() 52
__builtin_type() 52
__cplusplus 195
__DATE__ 194
__embedded_cplusplus 76, 195
__FILE__ 194
__fourbyteints__ 195
__ieeedoubles__ 196
__INTEL__ 196
__LINE__ 194
__MC68020__ 196
__MC68881__ 196
__MC68K__ 196
__MIPS__ 196
__MIPS_ISA2__ 196
__MIPS_ISA3__ 197
__MIPS_ISA4__ 197
__MWBROWSER__ 197
__MWERKS__ 197
__option(), preprocessor function 198
__powerc 197
__POWERPC__ 197
__PreInit__() 58
__profile__ 197
__STDC__ 194
__stdcall 41
__TIME__ 194

Numerics

3D 134, 135
__MC68020__ 196
__MC68881__ 196

A

__A5__ 195
a6frames pragma 90
Access Paths preference panel 31
Activate C++ Compiler option 62
address
    specifying for variable 40
align pragma 91
align_array_members pragma 92
Always Search User Paths option, Access Paths panel 32
always_inline pragma 93
AMD K6 3D 134, 135
anonymous structs 65
ANSI Keywords Only option 41
ANSI_strict pragma 94
arg_dep_lookup pragma 95
arguments
    unnamed 38
    unused 23
ARM Conformance option 63
ARM_conform 64
ARM_conform pragma 95
assignment, accidental 21
auto_inline pragma 44, 97

B

base classes
    protected 63
    referring to functions in 59
Be OS 85, 158
bool keyword 56, 64
bool pragma 97
by #pragma segment option 170

C

catch statement 54, 64, 66, 117
char 50
characters, multi-byte 42
check_header_flags pragma 98
CIncludes 32

code_seg pragma 99
code68020 pragma 99
code68881 pragma 100
commas, extra 24
comments, C++-styles 38
conditional operator 63
const_cast keyword 56
copy constructor 57
__cplusplus 195
cplusplus pragma 63, 101
cpp_extensions pragma 65, 102

D

d0_pointers pragma 103
__DATE__ 194
declaration
    of variable in statements 63
def_inherited pragma 61, 104
defer_codegen pragma 105
direct_destruction pragma 111
direct_to_som pragma 111
disable_registers pragma 112
DLL 27
dollar_identifiers pragma 113
Don't Inline option 44
Don't Reuse Strings option 46
dont_inline pragma 44, 113
dont_reuse_strings pragma 46, 114
dyanamic_cast keyword 162
dynamic_cast 67
dynamic_cast keyword 56

E

ecplusplus pragma 115
EIPC_EIPSW pragma 115
#else 39
empty declarations 20
Empty Declarations option 20
Enable Exception Handling option 64
#endif 39
Enum Always Int option 33
enumerated type 25
enumerated types 33
enumsalwaysint pragma 116
=
    accidental 21
    operator 57
errors
    and warnings 20
exception handling 64
exceptions pragma 117
.exp file 130
Expand Trigraphs option 42
explicit keyword 56
export pragma 118
Export Symbols option 130
Extended Error Checking option 24
extended_errorchecking pragma 25, 119
Extra Commas option 24

F

false keyword 56
far_code pragma 121
far_data pragma 122
far_strings pragma 122
far_vtables pragma 123
__FILE__ 194
for statement 21, 63
force_active pragma 124
__fourbyteints__ 195
fourbyteints pragma 124
fp_contract pragma 125
friend keyword 55

G

Global Register Allocation option 202
global_optimizer pragma 127

H

header files 30

I

identifiers 30
__ieeedoubles__ 196
IEEEdoubles pragma 128
if statement 21, 63

ignore_oldstyle pragma 128
Illegal Pragmas option 20
import pragma 129
include files, see header files
infinite loop 21
infinite loop, creating 21
inherited keyword 60, 104
inherited keyword 60
init_seg pragma 131
inline 105
inline_depth pragma 131
inline_intrinsics pragma 132
Inlining menu 43
integer formats 51
__INTEL__ 196
Intel MMX 135, 140
internal pragma 132

K

K6 3D 134, 135
keywords, additional 41

L

lib_export pragma 136
libraries 27
__LINE__ 194
long long 51
longlong 136
longlong_enums pragma 137

M

macros
    and # 38
macsbug pragma 137
main() 55
mangled names 27, 30
Map Newlines to CR option 49
__MC68020__ 196
__MC68881__ 196
__MC68K__ 196
member function pointer 65
microsoft_exceptions pragma 139
microsoft_RTTI pragma 139
__MIPS__ 196
__MIPS_ISA2__ 196
__MIPS_ISA4__ 197
__MIPS_ISA3__ 197
MMX 135, 140
mmx pragma 140
mpwc pragma 141
mpwc_newline pragma 49, 142
mpwc_relax pragma 50, 143
multi-byte characters 42
MultiMedia eXtensions 135, 140
mutable keyword 56
__MWBROWSER__ 197
__MWERKS__ 197

N

namespace keyword 56
near_code pragma 121

O

oldstyle_symbols pragma 137
once pragma 145
only_std_keywords pragma 42, 146
operator delete 62
operator new 62
operator= 57
opt_common_subs pragma 146
opt_dead_assignments pragma 147
opt_dead_code pragma 147
opt_lifetimes pragma 148
opt_loop_invariants pragma 148
opt_proagation pragma 149
opt_strength_reduction pragma 149
opt_unroll_loops pragma 150
opt_vectorize_loops pragma 150
optimization_level pragma 127
optimize_for_size pragma 151
__option(), preprocessor function 198
options align= pragma 91
Options Checking 198

P

pack pragma 152

parameter pragma 153
pascal keyword 41
pcrelstrings pragma 153
peephole pragma 154
pointer to member function 65
pointer types 50
pointers_in_A0 pragma 155
pointers_in_D0 pragma 155
Pool Strings option 45
pool_data pragma 156
pool_strings pragma 46, 157
pop pragma 158
Possible Errors option 21
__powerc 197
__POWERPC__ 197
pragma
    list of all 86
    scope 89
    syntax 88
#pragma statements
    illegal 20
#pragma statements 88
precompile_target pragma 158
Prefix File 32
__PreInit__() 58
preprocessor
    and # 38
__profile__ 197
profile pragma 159
protected base classes 63
prototypes
    requiring 47
push pragma 158

R

readonly_strings pragma 160
reinterpret_char keyword 56
Relaxed Pointer Type Rules option 50
Require Function Prototypes option 47
require_prototypes pragma 48, 161
return statement
    empty 25
    missing 24
RTTI 64, 162
RTTI option 64
RTTI pragma 162
Run-time type information 64, 162

S

scheduling pragma 163
Section Mappings panel, NEC V800 110
section pragma 163
segment pragma 170
side_effects pragma 171
simple class 57
simple_prepdump pragma 172
size_t 32
sizeof() operator 32
smart_code pragma 121
SOM Call Optimization pragma 172
SOMCallStyle pragma 173
SOMCheckEnvironment pragma 174
SOMClassVersion pragma 175
SOMMetaClass pragma 176
SOMRelaseOrder pragma 177
stack_cleanup pragma 177
static_cast keyword 56
static_inlines pragma 178
__STDC__ 194
string literals
    pooling 45
    reusing 46
structs
    anonymous 65
suppress_init_code pragma 179
switch statement 63
sym pragma 179
syspath pragma 180

T

template class statement 73
templates 70
__TIME__ 194
toc_data pragma 180
traceback pragma 182
Treat All Warnings as Errors option 20
trigraph characters 42

trigraphs pragma 42, 181
true keyword 56
try statement 54, 64, 66, 117
type_info 69
type-checking 50
typeid keyword 162
typeid keyword 56

U

unnamed arguments 38
unsigned char 50
unsigned_char pragma 182
Unused Arguments option 23
unused pragma 22, 23, 183
Unused Variables option 22
Use Unsigned Chars option 50
use_fp_instructions pragma 184
use_frame pragma 184
use_mask_registers pragma 184
using keyword 56

V

variables
    declaring by address 40
    unused 22
    volatile 33
virtual keyword 55
volatile variables 33

W

warn_emptydecl pragma 21, 185
warn_extracomma pragma 24, 186
warn_hidevirtual pragma 187
warn_illpragma pragma 20, 187
warn_possunwant pragma 22, 189
warn_structclass pragma 191
warn_unusedarg pragma 24, 191
warn_unusedvar pragma 23, 192
warning pragma 192
warning_errors pragma 20, 185
warnings
    as errors 20
wchar_type pragma 193
while statement 21, 63

CodeWarrior

C Compilers Reference

Credits

writing lead: Marc Paquette

other writers: Mark Anderson, Gene Backlin, BitHead, Jeff Mattson, Jim Trudeau

engineering: Mark Anderson, Bob Campbell, Ben Combee, Pascal Cleve, Rajeev Gulati, Andreas Hommel, Udi Kalekin, Michael Kahl, Bob Kushlis, John McEnerney, Fred Peterson, Laurent Visconti, Rhonda Wittels

frontline warriors: Richard Atwell, John C. Daub, Ron Liechty, John Roseborough, Joel Sumner, Jim Trudeau, L. Frank Turovich, CodeWarrior users everywhere

Guide to CodeWarrior Documentation

CodeWarrior documentation is modular, like the underlying tools. There are manuals for the core tools, languages, libraries, and targets. The exact documentation provided with any CodeWarrior product is tailored to the tools included with the product. Your product will not have every manual listed here. However, you will probably have additional manuals (not listed here) for utilities or other software specific to your product.

Core Documentation	
IDE User Guide	How to use the CodeWarrior IDE
Debugger User Guide	How to use the CodeWarrior debugger
CodeWarrior Core Tutorials	Step-by-step introduction to IDE components
Language/Compiler Documentation	
C Compilers Reference	Information on the C/C++ front-end compiler
Pascal Compilers Reference	Information on the Pascal front-end compiler
Error Reference	Comprehensive list of compiler/linker error messages, with many fixes
Pascal Language Reference	The Metrowerks implementation of ANS Pascal
Assembler Guide	Stand-alone assembler syntax
Command-Line Tools Reference	Command-line options for Mac OS and Be compilers
Plugin API Manual	The CodeWarrior plugin compiler/linker API
Library Documentation	
MSL C Reference	Function reference for the Metrowerks ANSI standard C library
MSL C++ Reference	Function reference for the Metrowerks ANSI standard C++ library
Pascal Library Reference	Function reference for the Metrowerks ANS Pascal library
MFC Reference	Reference for the Microsoft Foundation Classes for Win32
Win32 SDK Reference	Microsoft's Reference for the Win32 API
The PowerPlant Book	Introductory guide to the Metrowerks application framework for Mac OS
PowerPlant Advanced Topics	Advanced topics in PowerPlant programming for Mac OS
Targeting Manuals	
Targeting BeOS	How to use CodeWarrior to program for BeOS
Targeting Java VM	How to use CodeWarrior to program for the Java Virtual Machine
Targeting Mac OS	How to use CodeWarrior to program for Mac OS
Targeting MIPS	How to use CodeWarrior to program for MIPS embedded processors
Targeting NEC V810/830	How to use CodeWarrior to program for NEC V810/830 processors
Targeting Net Yaroze	How to use CodeWarrior to program for Net Yaroze game console
Targeting Nucleus	How to use CodeWarrior to program for the Nucleus RTOS
Targeting OS-9	How to use CodeWarrior to program for the OS-9 RTOS
Targeting Palm OS	How to use CodeWarrior to program for PalmPilot
Targeting PlayStation OS	How to use CodeWarrior to program for the PlayStation game console
Targeting PowerPC Embedded Systems	How to use CodeWarrior to program for PPC embedded processors
Targeting VxWorks	How to use CodeWarrior to program for the VxWorks RTOS
Targeting Win32	How to use CodeWarrior to program for Windows